National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

# An Extensible Query Optimizer Architecture for the TIGUKAT Objectbase Management System

By

Adriana Muñoz

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring 1994

National Library of Canada

Acquisitions and Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada

Direction des acquisitions et des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

*Your file Votre référence*

*Our file Notre référence*

**The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.**

**L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.**

**The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.**

**L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.**

ISBN 0-612-11305-1

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Adriana Muñoz

TITLE OF THESIS: An Extensible Query Optimizer Architecture for the TIGUKAT Objectbase Management System

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1994

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

(Signed) Adriana Muñoz

Permanent Address:
Calle 23 No. 36A - 12,
Bogota,
Colombia, South America

Date: January 31/94

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled *An Extensible Query Optimizer Architecture for the TIGUKAT Objectbase Management System* submitted by *Adriana Muñoz* in partial fulfillment of the requirements for the degree of Master of Science.

Supervisor: M.T. Özsu

Examiner: D. Szafron (Computing Science)

External: M. Rao ~~(Chemical Engineering)~~

Date: January 28/94

To my beloved friend Allan,
and to Mom and Dad

# Abstract

Objectbase Management Systems are expected to serve data management needs of a wide range of application domains with possibly different query optimization requirements, creating the need of extensibility in the query optimizer to be able to handle the diversity of those requirements.

This thesis describes the development of an extensible query optimizer architecture for the TIGUKAT Objectbase Management System, which has a uniform behavioral object model DBMS that represents every system component as a first-class object. Consistent with this philosophy, every component of the optimizer is modeled as a first-class object, providing the ultimate extensibility that the object-oriented paradigm offers. This thesis also describes how the optimizer components are modeled as extensions of the TIGUKAT type system.

# Acknowledgements

I would like to thank my supervisor Dr. M. Tamer Özsu for suggesting the topic of this research and for his invaluable guidance and support in writing this thesis. He provided an excellent research environment for my work.

Thanks are also due to Dr. Duane Szafron for his invaluable counsel about the object-oriented design of the extensible query optimizer architecture.

I would also like to thank the members of my examining committee, Dr. Ming Rao, Dr. Duane Szafron, and chair, Professor William Armstrong, for their valuable criticisms, comments and suggestions that helped me to produce the final version of this document.

I would like to acknowledge with appreciation all those who have helped me during my graduate studies. Thanks to my friends: Ian Parsons, and Kaladhar Voruganti, for the time they spent helping me to proofread this thesis. I am also thankful to my friends: Randal Peters, Yuri Leontiev, and the rest of the database group for the very constructive discussions that helped me to clarify my ideas in this field.

Finally, I wish to express my gratitude and appreciation to my beloved friend, Allan, my parents, sisters, brothers and the rest of my family for their constant support and encouragement throughout the completion of this dream.

# Contents


**1 Introduction** **1**

1.1 Query Processing Methodology . . . . . 2

1.1.1 Relational Query Processing Methodology . . . . . 2

1.1.2 Object-Oriented Query Processing Methodology . . . . . 2

1.2 Characterization of a Query Optimizer . . . . . 3

1.3 Scope of Thesis . . . . . 4

1.4 Organization of Thesis . . . . . 6

**2 Related Work** **7**

2.1 "Hard-wired" Object-Oriented Query Optimizers . . . . . 8

2.1.1 Orion . . . . . 8

2.1.2 ObjectStore . . . . . 10

2.1.3 $O_2$ . . . . . 10

2.1.4 Revelation . . . . . 11

2.2 Extensible Object-Oriented Query Optimizers . . . . . 12

2.2.1 Lanzelotte and Valduriez Proposal . . . . . 13

2.2.2 Open OODB Query Optimizer . . . . . 14

2.2.3 Epoq Architecture . . . . . 14

**3 TIGUKAT Overview** **16**

3.1 Object Model . . . . . 16

3.2 Query Model . . . . . 21

3.2.1 The Object Calculus . . . . . 23

3.2.2 The Object Algebra . . . . . 25

**4 Optimizer Architecture** **30**

4.1 Functions as Objects . . . . . 31

4.2 Queries as Objects . . . . . 33

4.2.1 Query Compilation . . . . . 36

4.3 Query Optimizer as an Object . . . . . 37

**5 Representation of Search Space** **40**

5.1 Object Algebra Processing Trees . . . . . 40

5.1.1 Execution Plan Generation . . . . . 44

5.1.2 Execution of the OAPT . . . . . 48

5.2 Algebraic Transformation Rules . . . . . 48

5.2.1 Specification of Algebraic Transformation Rules . . . . . 50

5.2.2 Algebraic Transformation Rules as Objects . . . . . 51

5.2.3 Rule Application . . . . . 52

**6 Modeling of Search Strategies** **61**

6.1 Search Strategies . . . . . 62

6.1.1 Algebraic Heuristic Search Strategies . . . . . 62

6.1.2 Cost-controlled Heuristic Search Strategies . . . . . 63

6.1.3 Enumerative Search Strategies . . . . . 64

6.1.4 Randomized Search Strategies . . . . . 64

6.2 Search Strategies as Objects . . . . . 65

6.3 Customizing the Search Strategy . . . . . 66

6.4 Extending the Search Strategy Component . . . . . 70

**7 Modeling of Cost Functions** **71**

7.1 Cost Model Functions . . . . . 73

7.1.1 Total Time Cost Function . . . . . 73

7.1.2 Response Time Cost Function . . . . . 74

7.2 Cost Model Functions as Objects . . . . . 74

7.2.1 Algebraic Node Cost Functions . . . . . 77
7.2.2 Execution of Cost Model Functions . . . . . 80

**8 Implementation of TIGUKAT Query Optimizer** **84**
8.1 Mapping of TIGUKAT Object Model to C++ . . . . . 87
8.1.1 Objects . . . . . 88
8.1.2 Types . . . . . 88
8.1.3 Type Hierarchy . . . . . 88
8.1.4 Collections and Classes . . . . . 89
8.1.5 Behaviors and Functions . . . . . 89
8.1.6 Behavioral and Implementation Inheritance . . . . . 90
8.2 Mapping of TIGUKAT Query Optimizer to C++ . . . . . 90
8.2.1 Search Space . . . . . 90
8.2.2 Search Strategy . . . . . 90
8.2.3 Cost Model Functions . . . . . 91

**9 Conclusions** **92**
9.1 Future Research . . . . . 93

**Bibliography** **94**

**A Walk Through the Optimizer Architecture by Example** **100**

**B Extensions to TIGUKAT Type System** **107**

# List of Tables


2.1 Summary of optimization features. . . . . 9
2.2 Feature comparison of extensible optimizers. . . . . 13

3.1 Behavior signatures pertaining to example specific types of Figure 3.2 . . . . . 22

4.1 Behavioral summary of `T_function` type. . . . . 34
4.2 Behavioral summary of `T_query` type. . . . . 35

5.1 Behavioral summary of `T_context` type. . . . . 42
5.2 Behavioral summary of `T_algOp` type. . . . . 43
5.3 Behavioral summary of `T_rule` type. . . . . 51
5.4 Behavioral summary of `T_algEqRule` type. . . . . 53
5.5 Behavioral summary of `T_formula` type. . . . . 56
5.6 Behavioral summary of operations on trees defined on `T_algOp` type. 57

6.1 Behavioral summary of `T_searchStrat` type. . . . . 65
6.2 Behavioral summary of `T_heurSS` type. . . . . 68
6.3 Extensibility behaviors for implementing heuristic and cost-controlled heuristic search strategies. . . . . 69
6.4 Behavioral summary of `T_CCHeurSS` type. . . . . 70

7.1 Behavioral summary of statistics defined on `T_collection` type. . 79

B.1 Behavioral summary of extended non-atomics primitive types for optimization purposes. . . . . 108

B.2 Behavioral summary of extended non-atomics primitive types for optimization purposes. . . . . 109
B.3 Behavioral summary of non-atomics types added to the primitive type system for optimization purposes. . . . . 110
B.4 Behavioral summary of non-atomics types added to the primitive type system for optimization purposes. . . . . 111

# List of Figures


3.1 Primitive type system . . . . . 17
3.2 Geographic Information System in TIGUKAT object model. . . . 20

4.1 Optimizer as part of the type system . . . . . 32

5.1 Collection (**L_implAlgOp**) of implementations for algebraic operators. . . . . 41
5.2 Tree shape of an OAPT. . . . . 42
5.3 Initial OAPT . . . . . 45
5.4 Transformed OAPT by using Rule 5.1 . . . . . 60

6.1 Type hierarchy for `T_searchStrat`. . . . . 67

7.1 Cost model functions as instances of `T_costFunc` . . . . . 75
7.2 Collection **L_implAlgOp** and class **C_costFuncAlgOp**. . . . . 77
7.3 OAPT annotated with algebraic node cost functions. . . . . 78
7.4 Execution of the total time cost model function *cf* on an OAPT. . 83

A.1 Object algebra processing tree . . . . . 106

# Chapter 1

# Introduction

One of the strengths of the relational database management systems (DBMSs) is the availability of declarative query capabilities which allow the users to specify "which" data they want retrieved from the database without having to specify "how" that data is accessed. The determination of the most efficient execution program to retrieve the requested data is taken over by the DBMS, in particular the query optimizer. The first generation objectbase management systems (OBMSs)[1] have been criticized [41] for their lack of declarative query capabilities. The newer systems and research prototypes have started to include these capabilities, but their optimization is still not very well understood.

These issues have been studied within the context of the TIGUKAT project[2]. TIGUKAT is an OBMS [32] under development at the University of Alberta. It has an extensible object model characterized by a purely behavioral semantics and a uniform approach to objects. The model is behavioral in that objects are accessed only by applying behaviors (which replace both the instance variables and the methods available in other object models) to objects. Behaviors are defined on types and their implementations are modeled as functions. Every concept, including types, classes, collections, and meta-information, is a first-

[1]The term *objectbase* is used instead of the traditionally used object-oriented database because the objects that it contains may include not only data, but also code.

[2]TIGUKAT (tee-goo-kat) is a term in the language of the Canadian Inuit people meaning"objects." The Canadian Inuits, commonly known as Eskimos, are native to Canada with an ancestry originating in the Arctic regions of the country.

class object. The uniformity of the object model extends to the query model, treating queries as first-class objects [36]. The query model consists of a calculus, an equivalent algebra and an SQL-based user query language.

## 1.1 Query Processing Methodology

TIGUKAT query processing methodology is similar to relational query processing, and follows the proposal for object-oriented query processing by Straube and Özsu [42]. There are differences in the query models, however, in that TIGUKAT query model is based on the uniform and behavioral features that its object model provides.

### 1.1.1 Relational Query Processing Methodology

In relational systems, query processing generally follows a straightforward methodology [34]: decomposition, optimization, and execution. The first step, *decomposition*, takes a query expressed in relational calculus, checks it for consistency, and translates it into an algebraic query. The query is represented as a processing tree where the leaf nodes represent base relations and the intermediate nodes correspond to relational operators. The second step, *optimization*, selects an "optimal" algebraic query among a family of equivalent queries. Optimal, in this context, refers to performance characteristics. The result of this step is a processing tree, each of whose nodes is annotated with the best execution algorithm for that particular algebraic operation (called the *access path*). Then, an execution plan is generated that mapps this processing tree to a set of storage system calls. The last step in the methodology executes the execution plan starting from the leaves up to the root.

### 1.1.2 Object-Oriented Query Processing Methodology

A query processing methodology for object-oriented databases is proposed by Straube and Özsu in [42] that follows closely that for relational systems. In this

methodology, queries, expressed in a declarative query language, are first normalized and converted to an equivalent algebra expression. The expression can be viewed as a tree whose nodes are algebra operators and whose leaves represent instances of classes in the database. In the next step, the algebra expression is checked for type consistency. Then, algebraic optimization is performed which consists of applying equivalence preserving rewrite rules to the type consistent algebra expression. These rules are defined as part of this work, but they are not implemented. The last step in query processing, called execution plan generation (EPG), produces an ordering of primitive low-level operations from the optimized algebra expression. This ordering is then passed to an object manager (OM). An important difference of this methodology from relational ones is that the encapsulation property of an object-oriented system may not allow the optimizer access to storage information that is needed for optimization. An EPG method is proposed in [43] to solve this problem by replacing each individual algebra operator from the transformed object algebra expression with a "best" subtree of object manager calls. These object manager calls are part of the set of low level object manipulation primitives that constitutes the interface to the object manager. This ordering of object manager calls is then passed to an object manager for further optimization and execution.

## 1.2 Characterization of a Query Optimizer

Query optimization can be modeled as an optimization problem whose solution is the choice of the "optimum" *state* in a *state space*. *States* are configurations of the objects relevant to the problem. Distinguished states are the initial and the goal states. A class of states defines the *search space*. *Actions* on states generate a set of successor states. These actions are controlled by the *search strategy*. In the case of a cost-controlled search strategy, a *cost function* applied to a state is used to measure the effects of the optimizer *actions*.

In algebraic query optimization, each state corresponds to an algebraic query

execution schedule represented as a processing tree (PT) [23]. A processing tree (PT) is a representation of a query, after it has been translated from a TQL statement to a corresponding algebraic expression. In this research, the processing trees are called *object algebra processing trees* (OAPT) to differentiate them from the traditional relational PTs.

The state space is a family of equivalent (in the sense of generating the same result) algebraic queries that can be generated by applying the transformation rules defined for the specific algebra. The goal is to move from one state to another using a *search strategy*, applying a *cost function* to each state in the search space and finding the one with the least cost.

Thus, to characterize a query optimizer four things need to be specified:

1. the states in the search space;
2. the transformation rules that generate the alternative query expressions which constitute the search space;
3. a search algorithm that allows one to move from one state to another in the search space; and
4. the cost function that is applied to each state.

## 1.3 Scope of Thesis

This thesis involves the development of a query optimizer architecture for TIGUKAT. There are a number of design considerations that were important in designing the query optimizer such as respecting the encapsulation of objects as provided by the object-oriented paradigm. The most important one is that the query optimizer should be extensible in every respect. There are a number of reasons for this insistence:

1. The optimization techniques for OBMSs are not fully developed and the alternatives are not completely understood. Thus, the techniques that are included in today's optimizers are likely to change as research results emerge.

It should be possible to easily incorporate these into the system. The problems that have been identified in the optimization of object-oriented queries and the techniques that have been proposed to solve them are surveyed in [30].

2. There is no consensus as to the set of object algebra operators that should be supported by these systems. Furthermore, OBMSs should allow application-specific algebra operators to be defined and managed by the system. In TIGUKAT, the definition of these operators is easy since they are defined as behaviors on collections. If the optimizer cannot deal with these operators uniformly, the purpose of defining them is defeated.

3. The application domains supported by the object-oriented technology have differing query processing/optimization requirements. Therefore, the optimizer design should support easy customization of the system.

This thesis describes the development of the architecture for the TIGUKAT query optimizer. This architecture allows every aspect of the algebraic query optimizer (search strategies, search space, algebraic transformation rules, and cost functions) to be extended. The extensibility approach presented in this thesis differs from those presented in other extensible query optimizers (that are discussed in Chapter 2) in that we use an object-oriented approach to extensibility that models each component of the query optimizer as an object. The incorporation of these optimizer components into the type system provides extensibility via the basic object-oriented principle of subtyping as it is described in the rest of this thesis. In addition, an implementation of the algebraic query optimizer is described. The work should provide an indication of the viability of using an object-oriented approach to provide extensibility to a query optimizer by modeling its components as objects.

## 1.4 Organization of Thesis

Chapter 2 discusses some of the previous work in object-oriented query optimization with particular emphasis on the extensible architectures. Chapter 3 summarizes the TIGUKAT object model and the query model, especially the algebra. The overall architecture of the TIGUKAT query optimizer is described in Chapter 4. Chapters 5, 6, and 7, respectively elaborate on the extensible representations of the search space, the search strategy, and the cost functions. Chapter 8 describes the implementation of the query optimizer using the C++ language. Conclusions and suggestions for further research are discussed in Chapter 9.

Appendix A presents an example query creation and optimization scenario to highlight how the various parts of the architecture interact. Appendix B presents the specification design of the extensions to the TIGUKAT type system for query optimization purposes.

# Chapter 2

# Related Work

Many existing OBMS query optimizers are either implemented as part of the object manager on top of a storage system, or they are implemented as client modules in a client-server architecture. In most cases, the four dimensions that characterize a query optimizer (states, search space, search strategy and cost function) are "hard-wired" into the query optimizer. These optimizers focus on specific techniques that solve particular optimization problems. For example, one highly visible problem in optimization of object-oriented queries is path expressions. Such expressions imply a navigation through objects to find the end of a path. Research in this area includes exploring indexing for paths (e.g., ObjectStore [31]), optimization in the presence of arbitrary methods along the path (e.g., Revelation project [12]), and the use of clustering and other storage information to determine path accesses (e.g., $O_2$ [4]). Some of these query optimizers are briefly discussed in Section 2.1.

Given that extensibility is a major goal of OBMSs, one would hope to develop an extensible optimizer that accommodates different search strategies, different algebra specifications with their different transformation rules, and different cost functions. Rule-based query optimizers [9] provide a limited amount of extensibility by allowing the definition of new transformation rules. However, they do not allow extensibility in other dimensions. In this chapter, some new proposals for extensibility that look promising for OBMSs are briefly highlighted in Section 2.2.

A more detailed discussion of these issues can be found in [33].

## 2.1 "Hard-wired" Object-Oriented Query Optimizers

In this section, the following query optimizers: Orion,ObjectStore, $O_2$, and Revelation, that focus on specific techniques to solve particular optimization problems are highlighted.

Table 2.1 summarizes the main optimization features that are supported by these query optimizers. These features were divided into two groups (that are shown separated by a double line in Table 2.1) as follows:

- The first group lists features about whether or not the query optimizer respects the encapsulation of objects as provided by the object-oriented paradigm, whether or not the optimizer uses *semantic* and/or *algebraic* transformation rules, and whether or not the optimizer factorizes common subexpressions that can appear in the query. *Semantic* transformations are transformation rules that exploit the semantics of the inheritance relationship that the object model provides. *Algebraic* transformations are transformation rules that create equivalent expressions based upon pattern matching and textual substitution.
- The second group illustrate features that are related to the optimization of path expressions.

For a given row $i$ and column $j$ in the Table 2.1, an entry with value **Yes** means that the query optimizer corresponding to the column $j$ supports the feature that appears in the row $i$. The value **No** means that the query optimizer does not support that feature, and the value — means that there is no information about whether that query optimizer supports that feature or not.

### 2.1.1 Orion

The Orion project [22] is the first attempt to define a query model for OBMSs that is consistent with object-oriented concepts. This model of queries is based

| Optimization features | Orion | ObjectStore | $O_2$ | Revelation |
|---|---|---|---|---|
| Respecting encapsulation | No | No | No | Yes |
| Semantic transformation rules | Yes | — | Yes | — |
| Algebraic transformation rules | No | No | Yes | Yes |
| Factorizing common subexpressions | — | — | Yes | — |
| Method optimization along paths | No | No | No | Yes |
| Indexing for paths | Yes | Yes | Yes | — |
| Clustering for path accesses | Yes | — | Yes | Yes |

Table 2.1: Summary of optimization features.

on the view that a query model may be defined as a subschema of the database *schema graph* which is an acyclic graph. The subschema that is called a *query graph* includes only the classes (and the hierarchies rooted at them) that participate in the predicates of the query. In Orion, a *class* is defined as a set of objects which share the same set of attributes and methods. A class provides the basis for query formulation. Therefore, a query must be formulated against a class or a class hierarchy rooted at that class. Orion query model extends the definition of relational operations such as joins and set operators (e.g., union) to be consistent with the semantics of object-oriented concepts.

In distributed Orion [19], the query optimizer uses algorithms to traverse trees and transformation rules that make use of type information, instead of algebraic transformation rules. This query optimizer takes into account the semantics of the class hierarchy and of complex objects. It defines complex object structures by reference links in a *part-of* hierarchy that is called a *class-composition* hierarchy. Orion uses path expressions to navigate through those complex object structures. The *class-composition* graphs are manipulated to find efficient traversals (forward and reverse), with access techniques such as indexes, applied to traversals. Orion query optimizer breaks the encapsulation of objects by accessing directly information about storage of objects (i.e. clustering) to determine path accesses.

### 2.1.2 ObjectStore

ObjectStore query model [31] is not uniform in the sense that a query not only returns objects, but also boolean and collections which are not considered objects. In this model, queries are based on selection of instances over a single collection. The query syntax for requesting a single element of a queried collection is different from the syntax for requesting a subset of that collection. Nested queries are allowed and they are existentially quantified. As far as we know, it does not yet implement join optimization. An essential feature of the language is that of predicates over paths. In ObjectStore, query processing involves the following steps: analysis, code generation, strategy selection, and execution. The first two steps are performed at compilation-time and the last two at run-time. Optimization focuses in comparison operators. ObjectStore has implemented *parametric optimization* that is also called dynamic plan selection. Using this technique, the query optimizer generates multiple execution plans at compile time and selects an "optimal" execution plan at run-time based on various system parameters and current database statistics. In this model, query optimization is based on information on paths and in the existence of indexes over paths that is obtained by accessing directly the storage structures. This violates the encapsulation of objects.

### 2.1.3 $O_2$

$O_2$ data model [2] is structural and non-uniform (i.e. values are not considered objects). The data model has classes and concrete types. The query language, $O_2$Query, is functional and is a subset of the programming language. It works in two modes [4]: a programming mode which respects encapsulation, and an interactive mode in which encapsulation can be broken by accessing directly the state of the objects through their structure. $O_2$ queries can return objects as well as values, but no dynamic creation of objects as a result of queries is allowed. Another feature of the language is that of providing both universal and existential quantifiers, while ObjectStore only provides existential and ORION only provides

universal quantifier. To tackle query optimization in $O_2$, Cluet and Delobel [4, 5] propose a formalism that

1. allows the integration of two existent query optimization approaches: algebraic equivalences similar to relational models and the concept of extension that is specific to object models,
2. allows an exhaustive factorization of duplicated subqueries, and
3. supports heuristics that make use of information on indexing and clustering in order to reduce the search space in the rewriting phase.

$O_2$ query optimizer does not consider evaluation costs nor the cost of accessing method code.

### 2.1.4 Revelation

The Revelation project [12] studies the optimization of queries respecting the encapsulated behavior of the objects. Its main features are the *Revealer* and the *Annotator* components that are preliminary phases to the optimization process and a special operator to assembly complex objects that is called *assembly*. The Revelation architecture has the following components: the Interpreter and the Schema Manager, the Revealer, the Annotator, the Optimizer and the Query Evaluator. The *Interpreter* takes an expression from the user and passes it to the Revealer component. In queries that contain operations on objects, the *Revealer* component is allowed to break the encapsulation of objects to find the method code for that operation which is translated into an algebraic formalism to allow the query *Optimizer* to manipulate it without having to break the encapsulation of objects. In order to expand a node, the Revealer makes requests to the *Annotator* which deduces the implementation method for the node by requesting the *Schema Manager* for information, and fills in this information for that node. This approach can have some problems with inheritance of methods and dynamic binding of methods because the *Annotator* may return a collection of values due to ambiguity arising from multiple implementations of a single operator. After

the expression tree is annotated, it is passed to the *Optimizer* for optimization and transformation to a query plan. The *Optimizer* is generated with the Volcano optimizer generator [13]. The Revelation query optimizer has limited extensibility provided by the Volcano optimizer generator. The algebra and the transformation rules can be extended without any problem, but more work is required in developing implementation rules which translates algebra trees into query plans, and in estimating costs for query plans. The Optimizer produces a query plan, that is passed to the *Query Evaluator*. The *Query Evaluator* is based on the extensible Volcano query execution software, which performs the plan execution. Volcano query execution software is extensible in the set of algorithms it employs. The *assembly* operator [20] is a special operator whose task is to avoid an *object-at-a-time* reading of the different components needed in the evaluation of a query involving complex objects. This operator is part of the query evaluator routines. A prototype for this query processor is in the process of implementation.

## 2.2 Extensible Object-Oriented Query Optimizers

In this section, the following new proposals for extensible query optimizers are highlighted: Lanzelotte and Valduriez proposal, Open OODB, and Epoq.

Table 2.2 summarizes the main optimization features that these optimizers support. These features were divided into three groups (that are shown separated by a double line in Table 2.2) as follows:

- the first group lists the different dimensions that can be extended in a query optimizer: search strategy, transformation rules, and cost model functions, as well as the algebra and its corresponding execution algorithms.
- the second group contains those features that were listed in the first group for the "hard-wired" query optimizers (that were presented in the previous section).
- the third group illustrates features that are related to the optimization of path expressions as those presented in the previous section.

| Optimization features | Lanzelotte et al. | Open OODB | Epoq |
|---|---|---|---|
| Extensibility of transformation rules | Yes | Yes | Yes |
| Extensibility of search strategy | Yes | No | Yes |
| Extensibility of cost model | — | Yes | Yes |
| Extensibility of algebra | Yes | Yes | — |
| Extensibility of execution algorithms | Yes | Yes | — |
| Respecting encapsulation | No | Yes | — |
| Semantic transformation rules | No | No | — |
| Algebraic transformation rules | Yes | Yes | Yes |
| Factorizing common subexpressions | Yes | — | — |
| Method optimization along paths | No | No | — |
| Indexing for paths | Yes | Yes | Yes |
| Clustering for path accesses | Yes | Yes | — |

Table 2.2: Feature comparison of extensible optimizers.

The entries in the Table 2.2 are interpreted in the same way as done for Table 2.1.

### 2.2.1 Lanzelotte and Valduriez Proposal

Lanzelotte and Valduriez [24] propose an extensible query optimizer by representing the search space and the search strategy components as objects and using the extensibility property that the object-oriented paradigm provides. However the other components are not modeled as objects and it is not mentioned how the cost component can be extended. This work considers the optimization of select, join and implicit join operator (which models path expressions). Executions plans are abstracted in terms of processing trees to have a more physical representation of the query. The leaf nodes of a processing tree are physical database sets (or subsets) and interior nodes represent joins that are explicit (i.e. a join predicate is given), implicit (i.e. a path from an object to an attribute of the object), or implicit with a path index. Their optimizer manipulates the processing tree by applying tree transformation rules. These transformation can be applied using deterministic and randomized search strategies based on the cost of the query. The transformations can result in path traversals that can start from any point in the path (not just the endpoints) and can be interleaved with other query opera-

tions [26]. This work is extended to handle recursive queries in [25] by introducing a fixpoint operator that handles the recursion. It makes use of a cost-controlled search strategy.

### 2.2.2 Open OODB Query Optimizer

The Open OODB project at Texas Instruments concentrates on the definition of an open architectural framework for OBMSs and on the description of the design space for these systems. The architecture of the system consists of three main kinds of components: *meta-architecture*, *support modules*, and *policy performers*. The meta-architecture houses all the mechanisms that provide the infrastructure for extending the operations of programming languages. It also defines the interface conventions to which policy modules must adhere. The support modules include address space managers (e.g., virtual memory, object repository), communications and translation managers for transferring objects among multiple address spaces, and a data dictionary providing name and type management facilities. The policy performer modules provide various database functionalities, one of which is query processing. Query processing in Open OODB [3] is largely influenced by the extensibility goals of the Open OODB project. The query module is an example of intra-module extensibility. The query optimizer, built using the Volcano optimizer generator [13], is extensible with respect to algebraic operators, logical transformation rules, execution algorithms, implementation rules (i.e., logical operator to execution algorithm mappings), cost estimation functions, and physical property enforcement functions (e.g., presence of objects in memory), but it is not extensible with respect to the search strategy. Volcano optimizer generates an exhaustive search algorithm (with some heuristics to prune the search space) that is hard-wired into the generated query optimizer.

### 2.2.3 Epoq Architecture

Quite a different approach to extensibility is proposed in the Epoq project [29], where the search space is divided into *regions*. Each region corresponds to

an equivalent family of query expressions that are reachable from one another. The regions do not have to be mutually exclusive and differ in the queries that they can manipulate, the control (search) strategy that they use or in the objectives that they want to achieve in query manipulation. For example, one region may have the objective of minimizing a cost function, while another region may attempt to put queries in some desirable form. Alternatively, one region may cover transformation rules that deal with simple select queries, while another region may deal with transformations for nested queries. Since a region incorporates a transformation strategy, it can be treated as the transformation of an input query to an equivalent output query. This is what allows movement between regions. There is a global control strategy to determine how the query optimizer moves from one region to another [28].

In this approach, the extensibility of the search space is accomplished at the region level since new regions can be defined to allow exploration of solutions that were previously unavailable. The extensibility of the cost function and the search strategy is encapsulated within each region (by, for example, hierarchically allowing different search strategies and transformations to use the same cost function). Since search strategies, transformations and optimizations can change from one region to another, the approach allows strategies to be changed within a single query. However, the feasibility and effectiveness of this approach have not yet been verified.

# Chapter 3

# TIGUKAT Overview

In this chapter an overview of TIGUKAT is given. Section 3.1 outlines the main characteristics of the TIGUKAT object model, including a description of such concepts as objects, types, classes, behaviors, functions, and the relationships among them. Section 3.2 describes the TIGUKAT query model which provides the declarative query facilities to the object model. Two formal languages are defined: an object calculus and an equivalent object algebra.

## 3.1 Object Model

The TIGUKAT object model [35, 39] is defined *behaviorally* with a uniform semantics. The model is *behavioral* in the sense that the access and manipulation of objects is restricted to the application of behaviors (operations) upon objects. The model is *uniform* in that every concept within the model has the status of a *first-class object*. An object is a fundamental concept in TIGUKAT. Every component of information, including its semantics, is uniformly represented by objects in TIGUKAT. This means that at the most basic level, every expressible element in the model incorporates at least the semantics of our primitive notion for "object".

The model defines a number of primitive objects which include: *atomic entities* (such as reals, integers, strings, characters, etc.); *types* for defining and

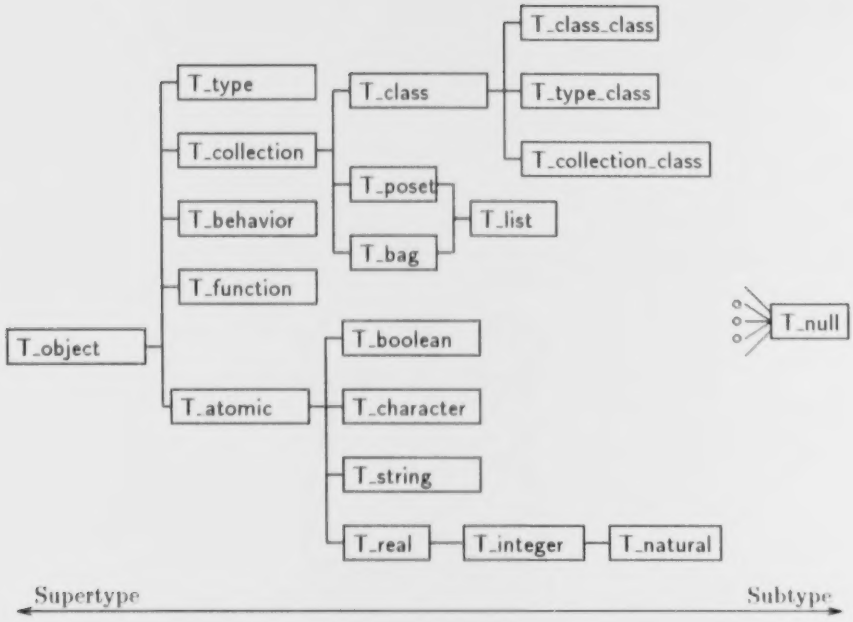


Figure 3.1: Primitive type system

structuring features of common objects; *behaviors* for specifying the semantics of the operations which may be performed on objects; *functions* for specifying the implementation of behaviors over various types forming the support mechanism for overloading and late binding; *classes* for the automatic classification of objects based on type; and *collections* for supporting general heterogeneous user-definable groupings of objects.

The primitive type system is shown in Figure 3.1 with the `T_object` type as the root of the lattice and the `T_null` type as the base. `T_null` binds the lattice from the bottom. It is a subtype of every other type in the system. `T_null` is introduced in the model to provide an object which can be returned by behaviors that have no result.

Objects are defined as *(identity, state)* pairs where *identity* represents a unique, immutable system managed object identity, and *state* represents the information carried by the object. Thus, the model supports *strong object identity* [21],

meaning that every object has a unique existence within the model and is distinguishable from every other object. On the other hand, the *state* of an object *encapsulates* the information carried by that object. Conceptually, every object is a *composite* object in TIGUKAT meaning that every object has references to other objects.

There is a separation of means for defining the characteristics of object (i.e., a *type*) from the mechanism for grouping of instances of a particular type (i.e., *class*). A *type* specifies behaviors. It encapsulates hidden implementation and state for all objects that are created by using the type as a template. The set of behaviors defined by a type is referred to as a set of *native* behaviors, and it describes the *interface* of the objects of that type. Types are organized into a lattice structure using the notion of *subtyping*. TIGUKAT supports *multiple inheritance*, meaning that one type can be an immediate subtype of several other types.

A *class* ties together the notion of *type* and *object instance*. A *class* is responsible for managing all instances that are created by using a specific type as a template. Objects of a particular type cannot exists without an associated class and every class is uniquely associated with a single type. Object creation occurs only through a class using its associated type as a template for the creation.

A *collection* is another grouping construct in TIGUKAT. It is defined as a general user-definable construct. It is similar to a *class* in that it also represents an extent of objects, but it differs in the following respects. First, no object creation can occur through a collection; object creation occurs only through classes. Second, an object may exist in any number of collections, but it is a member of only one class. Third, the management of classes is *implicit* in that the system automatically maintains classes based on the subtype lattice, whereas the management of collections is *explicit*, meaning that the user is responsible for their extents. Finally, a class groups the entire extension of a single type (*shallow extent*), along with the extensions of all its subtypes (*deep extent*). Therefore, the elements of a class are homogeneous up to inclusion polymorphism. On the other

hand, a collection may be heterogeneous in the sense that it can contain objects which may be of different types.

The subtypes of `T_class` namely, `T_class-class`, `T_type-class`, and `T_collection-class`, are part of the *meta* system. Their placement within the type system itself directly supports uniformity of the model. A full explanation of these types can be found in [38].

Two other fundamental notions of TIGUKAT are *behaviors* and *functions* that implement the behaviors. In the same way that an object's specification (types) is separated from the grouping of its elements (classes), the definition of a behavior is separated from its possible implementations (function/methods).

The semantics of each operation on an object is specified by a behavior defined on its type. A function implements the semantics of each behavior. The implementation of a particular behavior may vary over the types which support it. Nevertheless, the semantics of the behavior remains constant and unique over all types supporting that behavior. There are two kinds of implementations for behaviors. A *computed function* consists of runtime calls to executable code. A *stored function* is a reference to an existing object in the objectbase. The uniformity of TIGUKAT considers each behavior application as the invocation of a function, regardless of whether the function is stored or computed.

The following example illustrates a geographic information system in the TIGUKAT object model. This example, taken from [39], will be used as a running example throughout this thesis.

**Example 3.1** Object-orientation is intended to serve many application areas requiring advanced data representation and manipulation. A geographic information system (GIS) [1, 47] has been selected as an example to illustrate the practicality of the concepts introduced and to assist in clarifying their semantics. A GIS was chosen because it is among the application domains which can potentially benefit from the advanced features offered by object-oriented technology. Specifically, a GIS requires the following capabilities:

1. management of persistent and transient data,

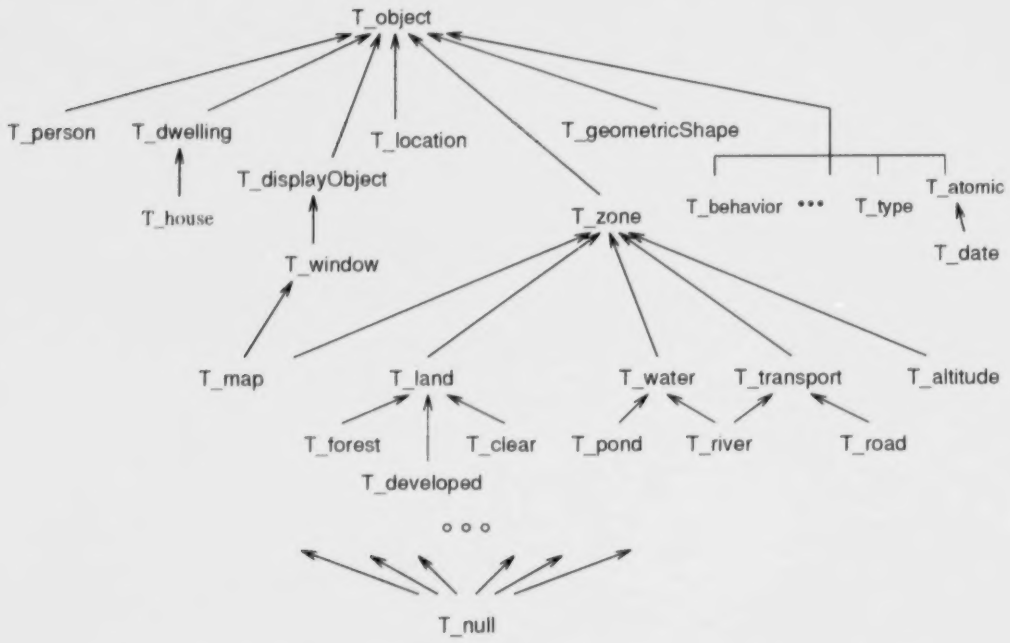


Figure 3.2: Geographic Information System in TIGUKAT object model.

2. management of large quantities of diverse data types and dynamic evolution of types,

3. a seamless integration of sophisticated computer graphic images with complex structured attribute data,

4. handling of large volumes of data and performing extensive numerical tabulations on data,

5. management of differing views of data, and

6. the ability to efficiently answer a variety of ad hoc queries.

A type lattice for a simplified GIS is given in Figure 3.2. The example is sufficiently complex to illustrate the advanced functionality of the query model we present, yet simple enough to be understandable without an elaborate discussion. The example includes the root types of the various sub-lattices from the primitive type system in Figure 3.1 to illustrate their relative position in an extended

application lattice. The additional types defined by the GIS example include:

1. Abstract types for representing information on people and their dwellings. These include the types `T_person`, `T_dwelling` and `T_house`.

2. Geographic types to store information about the locations of dwellings and their surrounding areas. These include the type `T_location`, the type `T_zone` along with its subtypes which categorize the various zones of a geographic area, and the type `T_map` which defines a collection of zones suitable for displaying in a window.

3. Displayable types for presenting information on a graphical device. These include the types `T_displayObject` and `T_window` which are application independent and the type `T_map` which is the only GIS application specific object that can be displayed.

4. A type `T_geometricShape` that defines the geometric shape of the regions representing the various zones. For our purposes we will only use this general type, but in more practical applications this type would be further specialized into subtypes representing polygons, polygons with holes, rectangles, squares, splines and so on.

## 3.2 Query Model

The complete uniform behavioral object model has formed the basis for an object query model that includes a complete algebra with an equivalent object calculus definition. An underlying characteristic of the TIGUKAT query model is that it is a direct extension to the object model. In other words, it is defined by type and behavior extensions to the primitive model.

The subsequent sections summarize the formal languages defined for the TIGUKAT query model. The full specification of the query model is given in [36, 37]. In the following sections, we first present the object calculus, and then the object algebra. The calculus has first-order semantics. Its logical foundation

| Type | Signatures | |
|---|---|---|
| T_location | *B_latitude*: | T_real |
| | *B_longitude*: | T_real |
| T_displayObject | *B_display*: | T_displayObject |
| T_window | *B_resize*: | T_window |
| | *B_drag*: | T_window |
| T_geometricShape | | |
| T_zone | *B_title*: | T_string |
| | *B_origin*: | T_location |
| | *B_region*: | T_geometricShape |
| | *B_area*: | T_real |
| | *B_proximity*: | T_zone → T_real |
| T_map | *B_resolution*: | T_real |
| | *B_orientation*: | T_real |
| | *B_zones*: | T_collection⟨T_zone⟩ |
| T_land | *B_value*: | T_real |
| T_water | *B_volume*: | T_real |
| T_transport | *B_efficiency*: | T_real |
| T_altitude | *B_low*: | T_integer |
| | *B_high*: | T_integer |
| T_person | *B_name*: | T_string |
| | *B_birthDate*: | T_date |
| | *B_age*: | T_natural |
| | *B_residence*: | T_dwelling |
| | *B_spouse*: | T_person |
| | *B_children*: | T_person → T_collection⟨T_person⟩ |
| T_dwelling | *B_address*: | T_string |
| | *B_inZone*: | T_land |
| T_house | *B_inZone*: | T_developed[a] |
| | *B_mortgage*: | T_real |

[a] Behavior was refined from supertype T_dwelling.

Table 3.1: Behavior signatures pertaining to example specific types of Figure 3.2

includes a definition of atoms, well-formed formulas, and a function symbol which incorporates the behavioral nature of the object model. This allows the use of very general path expressions in the calculus. The safety of the calculus is based on the *evaluable* class of queries defined in [10]. The object algebra consists of target-preserving and target-creating algebraic operators and is proven to be equivalent to the object calculus.

### 3.2.1 The Object Calculus

The alphabet of the calculus consists of object constants $(a, b, c, d)$, object variables $(o, p, q, u, v, x, y, z)$, monadic predicates $(C, P, Q)$, dyadic predicates $(=, \neg, \in, \notin)$, an $n-$ary predicate $(Eval)$, a function symbol $(\beta)$, logical connectives $(\exists, \forall, \wedge, \vee, \neg)$, and delimiters (&, (),).

Atoms are the building blocks of calculus expressions. The *atoms* of the calculus consist of the following:

**Range Atom:** $C(s)$ is called a range atom for $s$ where $C$ corresponds to a unary predicate representing a collection and $s$ denotes a term. A range atom asserts true if and only if $s$ denotes an object in collection $C$. When $C$ defines a class, $C^+(s)$ is true if and only if $s$ denotes an object in the *shallow extent* of class $C$.

**Equality Atom:** $s = t$ is a built-in predicate called an *equality atom* where $s$ and $t$ are terms. The predicate asserts true if and only if the object denoted by $s$ is object identity equal to the object denoted by $t$.

**Membership Atom:** $s \in t$ is a built-in predicate called a *membership atom* where $s$ and $t$ are terms, and $t$ denotes a collection. The predicate asserts true, if and only if the object denoted by $s$ is an element of the collection denoted by $t$.

**Generating Atom:** Any equality atom of the form $o = t$ or membership atom $o \in t$, where $o$ is an object variable and $t$ is an appropriate term for the

atom in which $o$ does not appear, is called a *generating atom* for $o$. That means that the object denotation for $o$ can be generated from $t$.

The *ground atom* is an atom that contains only ground terms.

From atoms, *well-formed formulas* (WFFs) are built to construct the declarative calculus expressions of the language. WFFs are defined recursively from atoms in the usual way [6, 48] using the connectives $\wedge, \vee, \neg$ and the quantifiers $\exists$ and $\forall$.

A *target-preserving* query is an object calculus expression of the form $\{t|\phi(o)\}$ where $t$ is a term consisting of a single object variable or an object variable indexed by a list of behaviors, $\phi$ is a WFF, and $o$ is exactly the variable in $t$ and it is the only free variable referenced in $\phi$. Indexed object variables are of the form $o[\beta]$ where $\beta$ is a set of behaviors defined on the type of variable $o$. The semantics of this construct is to project over the behaviors in $\beta$ for $o$, meaning that after the operation only the behaviors given in $\beta$ will be applicable to $o$. A *target-creating* query is of the form $\{t_1, ..., t_n|\phi(o_1, ..., o_n\}$ which is simply a generalization of the target-preserving kind by allowing multiple target terms $t_1, ..., t_n$ over the multiple variables $o_1, ..., o_n$. The result of such a query is a collection of new object lists created from the cartesian product over ranges of variables $o_1, ..., o_n$ by following the selection using $\phi(o_1, ..., o_n)$.

**Example 3.2** Target-preserving query: *Return all zones that are part of the same map. Project the result over B_title and B_area.*

$$\{o[B\_title, B\_area]|\exists p(\mathbf{C\_map}(p) \wedge o \in p.B\_zones)\}$$

$o$ is a free variable generated by the generating atom: $o \in p.B\_zones$, and $t = o[B\_title, B\_area]$ is a target variable in form of the index variable.

**Example 3.3** Target-creating query: *Return all the people and their spouses such that both of them are older than 65 years old*

$$\{p, q|\mathbf{C\_person}(p) \wedge q = p.B\_spouse \wedge p.B\_age > 65 \wedge q.B\_age > 65\}$$

Since, there are two target variables in the target list, this is an example of a target-creating query.

### 3.2.2 The Object Algebra

The operands and results of the object algebraic operators are typed collections of objects. The algebra maintains the closure property since the results of any operator may be used as an operand of another. The object algebra defines both *target-preserving* and *target-creating* operators. The target- preserving operators are defined as follows:

**Set Operations:** The typical set **union, difference** and **intersection** operators are defined.

**Select** (denoted $P\sigma_{[F]} < Q_1, ..., Q_n >$): Select is a higher order predicate that accepts the predicate $F$, and the n+1-ary collection $P, Q_1, ..., Q_n$ as arguments. The result collection contains objects from $P$ corresponding to the $p$ components of each permutation $< p, q_1, ..., q_n >$ that satisfies $F$.

**Map** (denoted $Q_1 \gg_{mop} < Q_2, ...Q_n >$): where *mop* is a mop function [37] over the elements of collections $Q_1, Q_2, ..., Q_n$, meaning it expects arguments $q_1, q_2, ..., q_n$ and they are type consistent with the membership types of the collections. For each permutation of objects $< q_1, q_2, ..., q_n >$ form from the elements of the argument collections $mop(q_1, q_2, ..., q_n)$ is applied and the resulting object is included in the result collection.

**Project** (denoted $P\Pi_\beta$): where $P$ is a collection and $\beta$ is a behavioral projection set with the restriction that it is a subset of the behaviors defined on the membership type of $P$. The $\beta$ collection is automatically unioned with the behaviors of type `T_object` in order to ensure consistency. The result collection contains objects of $P$, but with the membership type coinciding with the behavior specification of $\beta$.

The full object algebra includes target-creating operators in order to provide necessary object formation operators. The result of these operations is a collec-

tion of new objects that are object identity distinguishable from the ones in the argument collection. The primary target-creating operator is *product*:

**Product** (denoted $P \times Q$): Product produces a collection containing product objects created from each permutation $< p, q >$ such that the left component is an object from $P$ and the right component is an object from $Q$. Product may initiate the creation of a new type along with a new class to maintain the product objects.

The above collection of operators form the *primitive* algebra. They are fundamental in supporting the expressive power of the calculus and other expressions can be defined in terms of them. The following operators are added to the primitive algebra in order to provide functionality, and increase the expressive power.

**Join** (denoted $P \bowtie_{[F]} < Q_1, ..., Q_n >$): where $n \geq 1$. Join produces a collection containing product objects created from each permutation $< p, q_1, ..., q_n >$ that satisfies $F$.

**Generate Join** (denoted $Q_1 \gamma^{o}_{[g]} < Q_2, ..., Q_n >$): where $g$ is a generating atom of the form $o\theta < \vec{q} > .\vec{b}$ (where $\theta$ is one of '=' or '$\in$') over the elements of collections $Q_1, Q_2, ..., Q_n$. Generate join produces a collection of product objects created from each permutation of the $q_i$'s and extended by an object $o$ in the following way. If $\theta$ is '=', the result contains product objects of the form $< q_1, ..., q_n, < q_1, ..., q_n > .\vec{b} >$ for each permutation of the $q_i$'s. If $\theta$ is $\in$, the result contains product objects of the form $< q_1, ..., q_n, o >$ for each permutation of the $q_i$'s and $o \in < q_1, ..., q_n > .\vec{b}$.

**Reduce** (denoted $P_{\vec{p}} \triangle_{\vec{o}}$): where $P$ is a collection of product objects $\vec{p}$, and $\vec{o}$ is a list representing symbolic reference to the component of the product. The reduce operator has the effect of discarding the $\vec{o}$ components of the objects in $P$. That is, product objects of the form $< p_1, ..., p_i, \vec{o}, p_{i+1}, ..., p_n >$ are mapped to $< p_1, ..., p_i, p_{i+1}, ..., p_n >$.

**Collapse** (denoted $P \Downarrow$): Collapse is a unary operator which accepts a collection of collections $P$ as an argument and it produces the extended union of the

collections in $P$.

It is assumed that formulas that are part of operators in the object algebra (i.e. formula F in the select operator) are propositional formulas that are expressed in a conjunctive normal form (CNF) with the restriction that none of its atoms can have the logical connector $\neg$ (negation). A propositional form is defined in a similar way as a *well-formed formula* was defined in Section 3.2.1, but it is not quantified[1].

The following examples illustrate possible queries on the GIS defined in Example 3.1. Every query is given in the form of an English sentence, then it is expressed in the object calculus which is followed by the equivalent algebraic expression. In the algebraic expressions, operand collections are subscripted by the variable that ranges over them. If the operand consists of product objects, the variables that make up the components of these objects are listed. The indexed variables are used as a symbolic reference to the elements of the collection as described in this section. Furthermore, the arithmetic notation for operations like *o.greaterthan(p)* and *o.elementof(p)* (i.e. $o > p$ and $o \in p$, respectively) is used instead of boolean behavioral specification (Bspec) equivalents. The execution of an algebraic expression is from left-to-right, except that parenthesized expressions are executed first.

**Example 3.4** Return land zones valued over $100,000 or covering an area over 1000 units.

Calculus:

$$\{\ o \mid \mathbf{C\_land}(o) \wedge (o.B\_value > 100000 \vee o.B\_area > 1000)\}$$

Algebra:

$$\mathbf{C\_land}_o\ \sigma_{[o.B\_value > 100000\ \vee\ o.B\_area > 1000]}$$

[1]Quantifiers $\exists$ and $\forall$ does not appear in a propositional formula

**Example 3.5** Return all zones that have people living in them (the zones are generated from person objects).

Calculus:

$$\{\ o \mid \exists q(\mathbf{C\_person}(q) \wedge o = q.B\_residence.B\_inzone)\}$$

Algebra:

$$\left(\mathbf{C\_person}_q\ \gamma^{o}_{o=q.B\_residence.B\_inzone}\ \right)_{o,q}\ \triangle_q$$

**Example 3.6** Return the maps with areas where citizens over 65 years of age live.

Calculus:

$$\{\ o \mid \mathbf{C\_map}(o)\ \wedge\ \exists p(\mathbf{C\_person}(p)\ \wedge\ \exists q(\mathbf{C\_dwelling}(q)$$
$$\wedge\ p.B\_age \geq 65\ \wedge\ q = p.B\_residence\ \wedge\ q.B\_inzone \in o.B\_zones))\}$$

Algebra:

$$\left(\mathbf{C\_map}_o \bowtie_{F_1} \langle \mathbf{C\_dwelling}_q,\ \left(\mathbf{C\_person}_p\ \sigma_{F_2}\ \right)_p \rangle\right)_{o,q,p}\ \triangle_{p,q}$$

where $F_1$ is the predicate $(q = p.B\_residence\ \wedge\ q.B\_inzone \in o.B\_zones)$ and $F_2$ is the predicate $(p.B\_age \geq 65)$

**Example 3.7** Return the dollar values of the zones that people live in.

Calculus:

$$\{\ o \mid \exists p(\mathbf{C\_person}(p) \wedge o = p.B\_residence.B\_inzone.B\_value)\}.$$

Algebra:

$$\left(\mathbf{C\_person}_p\ \gamma^{o}_{o=p.B\_residence.B\_inzone.B\_value}\ \right)_{p,o}\ \triangle_p$$

Note that this has a simpler form using the map operator as follows:

$$\mathbf{C\_person}_p \gg_{p.B\_residence.B\_inzone.B\_value}$$

**Example 3.8** Return the zones that are part of some map and are within 10 units from water. Project the result over *B_title* and *B_area*.

Calculus:

$$\{\ o[B\_title, B\_area] \mid \exists p \exists q(\mathbf{C\_map}(p) \wedge \mathbf{C\_water}(q)$$

$$\wedge o \in p.B\_zones \wedge o.B\_proximity(q) < 10)\}.$$

Algebra:

$$\left(\left(\mathbf{C\_map}_p\ \gamma^{o}_{F_1}\ \right)_{p,o} \bowtie_{F_2} \mathbf{C\_water}_q\right)_{p,o,q} \triangle_{q,p}\ \Pi_{B\_title.B\_name}$$

where $F_1$ is a generating atom ($o \in p.B\_zones$)
and $F_2$ is a predicate ($o.B\_proximity(q) < 10$)

**Example 3.9** Return pairs consisting of a person and the title of a map such that the person's dwelling is in the map.

Calculus:

$$\{p, o \mid \exists q(\mathbf{C\_person}(p) \wedge \mathbf{C\_map}(q)$$
$$\wedge\ o = q.B\_title \wedge p.B\_residence.B\_inZone \in q.B\_zones)\}$$

Algebra:

$$\left(\mathbf{C\_person}_p \bowtie_F \left(\mathbf{C\_map}_q\ \gamma^{o}_{o=q.B\_title}\ \right)_{q,o}\right)_{p,q,o} \triangle_q$$

where $F$ is a predicate ($p.B\_residence.B\_inZone \in q.B\_zones$)

# Chapter 4

# Optimizer Architecture

TIGUKAT query optimizer models the search space, the transformation rules, the search strategy, and the cost function independently from each other, allowing extensibility in all dimensions.

All of the system concepts are represented as objects along the lines of [24], but differs from it in that the components of the TIGUKAT query optimizer are uniformly described in terms of the object model itself. The algebraic operators are modeled as objects, specifically as behaviors over the `T_collection` type. In the type lattice, they appear as instances of type `T_algebra` which is a subtype of `T_behavior` (instances are shown as rounded-corner rectangles, drawn with dashed lines in Figure 4.1 while types are shown as simple rectangles). In addition, the four components of the query optimizer are modeled as objects as listed below (see Figure 4.1),

1. the states in the search space that are modeled as query expressions which are represented by OAPTs (OAPTS are objects of type `T_algOp`);
2. the actions that generate the alternative query expressions. These actions are modeled as transformation rules (which are objects of type `T_algEqRule`);
3. a search algorithm that allows one to move from one state to another in the search space (search strategies are objects of type `T_searchStrat`); and

4. the cost function that is applied to each state (that are objects of type `T_costFunc`).

The incorporation of these components of the optimizer into the type system provides extensibility via the basic object-oriented principle of subtyping and specialization. In the following three chapters the modeling of the components of the optimizer as part of TIGUKAT's type system is discussed.

Because most of the extensions to the primitive type lattice for query optimization purposes are done by subtyping `T_function` (see Figure 4.1), a description of functions as objects is relevant to this discussion (presented in Section 4.1).

In TIGUKAT, queries have the status of *first-class objects* and inherit all the behaviors and semantics of objects. Since queries are so fundamental to this discussion, a description of modeling queries as objects is given in Section 4.2.

A unique feature of TIGUKAT query optimizer is that not only its components, but the optimizer itself is modeled as an object. This is discussed in Section 4.3.

## 4.1 Functions as Objects

In TIGUKAT object model, functions implement the semantics of behaviors. They are modeled by objects of type `T_function` and they have the status of a *first-class* object as illustrated in Figure 4.1. Since they are so fundamental to this discussion, the most important behaviors defined on `T_function` are listed in Table 4.1[1].

The major benefits of modeling functions as objects in TIGUKAT object model are the following:

1. Functions are *first-class* objects, so they support the uniform semantics of objects. They are maintained within the objectbase and are accessible

[1]For every behavior $b$ listed in the table, a functional application notation $b(o)(p)$ is used, where the first parameter $o$ corresponds to $b$'s receiver object and $p$ corresponds to $b$'s arguments. This applies to all the tables that summarize the native behaviors of an specific type through this thesis.

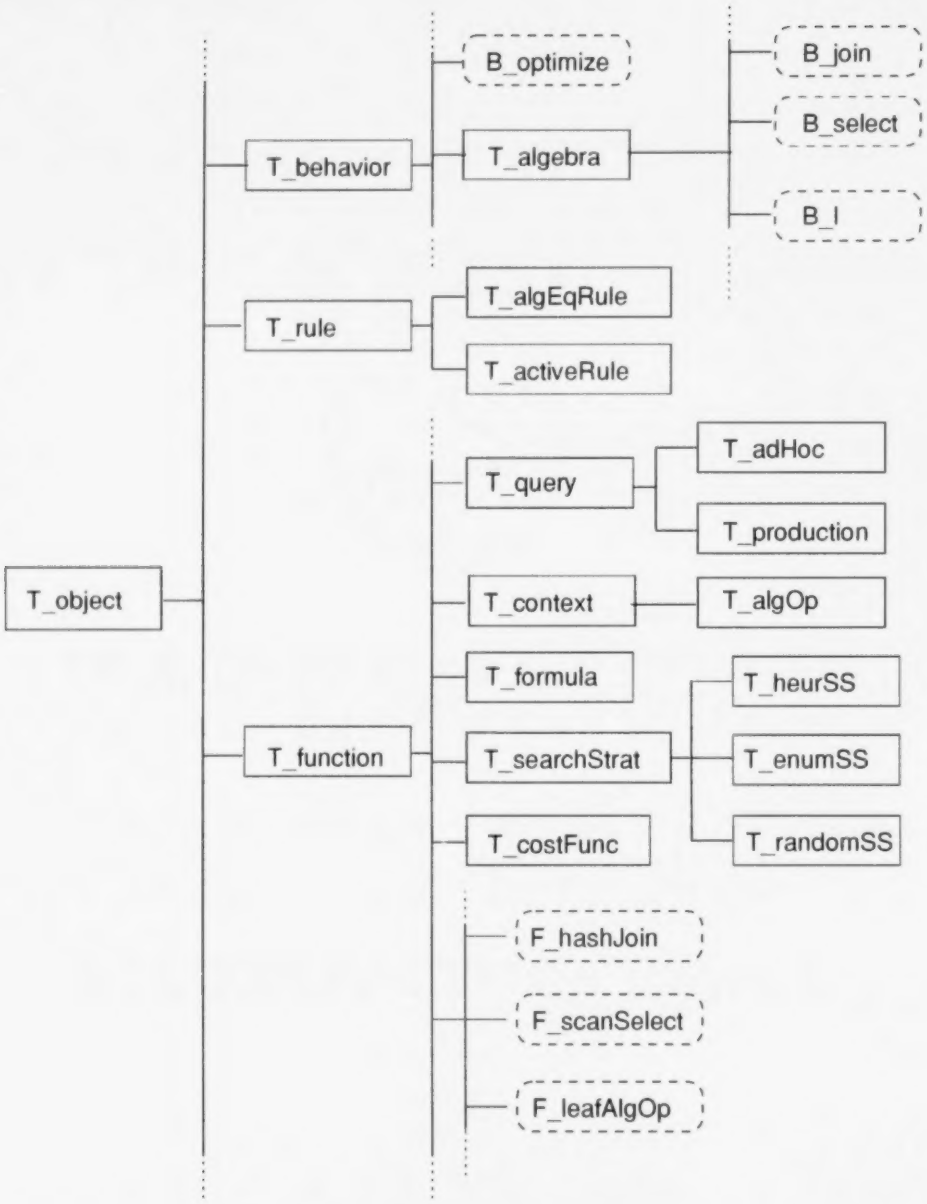


Figure 4.1: Optimizer as part of the type system

through the behavioral paradigm of the object model.

2. Since functions are objects, behaviors can be applied to them. This is useful in defining a uniform extensible query optimizer.

3. Functions are uniformly integrated with the operational semantics of the model and thus, functions are used as implementations of behaviors (i.e., the result of applying a behavior to an object triggers the execution of a function).

4. Functions as objects allow us to have different implementations for each object by plugging in a (stored) behavior a reference to a function object which can be executed later on. In contrast, behaviors restrict us to have one implementation per type. This is a fundamental feature to be able to model some of the components of the optimizer as objects as discussed in the following three chapters.

## 4.2 Queries as Objects

An identifying characteristic of the TIGUKAT query model is that it is a direct extension to the object model. In other words, it is defined by type and behavioral extensions to the primitive model. The query model defines a type `T_query` as a subtype of `T_function` in the primitive type system as illustrated in Figure 4.1. This means that queries have the status of *first-class objects* and inherit all the behaviors and semantics of objects. Moreover, queries are functions so they can be used as implementations of behaviors, can be compiled, and can be executed.

Since `T_query` is a subtype of `T_function`, it inherits all of the behaviors discussed in Section 4.1 and defines new ones. For a query, *B_compile* is redefined to include translating the query statement into an algebra tree, optimizing it and generating an execution plan (a more detailed description of the compilation process is given in section 4.2.1). Similarly, *B_execute* is redefined to mean that

| Native Behaviors | Semantics |
|---|---|
| *B_name*(*o*) | Returns the name of the function object *o*. |
| *B_argTypes*(*o*) | Returns a list of types which denote the types and ordering of the argument objects for the function object *o*. |
| *B_resultType*(*o*) | Returns the result type of the function object *o*. |
| *B_comments*(*o*) | Returns the comments that document the function object *o*. |
| *B_source*(*o*) | Returns the source code of the function object *o*. |
| *B_compile*(*o*) | Compiles the function object *o* and returns an executable which is also returned by *B_executable* below. |
| *B_execute*(*o*)(*p*) | Executes the function object *o* using the objects in the list *p* as arguments and returns a result object. One of the requirements is that the list of arguments in *p* must be compatible with the argument type list for the function. |
| *B_executable*(*o*) | Returns the executable code of the function argument object *o*. |
| *B_costFunction*(*o*)(*p*) | Returns a cost function object whose execution returns an estimated cost of executing the function *o* with arguments *p*. At the moment, this is defaulted to a cost function that returns unit cost for all functions other than those of type `T_algOp`. |
| *B_cost*(*o*)(*p*) | Executes the cost function object that is returned by *B_costFunction* and returns the estimated cost that results of this execution. |

Table 4.1: Behavioral summary of `T_function` type.

the execution plan is submitted to the storage manager for processing. There is the capability to store the result of query execution so that it is returned the next time the query is posed without re-execution. These details are not in the scope of this thesis because the execution of a query is not considered in this research. Furthermore, *B_costFunction*(*o*)(*p*) is redefined to return the cost function that the optimizer (*B_optimize*) uses when the search strategy (*B_searchStrat*) is cost-controlled. Otherwise, it returns `null`. It must be determined externally, before *B_optimize* is applied (i.e. when the query object *o* is created).

In addition to the behaviors it inherits from `T_function`, `T_query` has the native behaviors that are listed in Table 4.2.

Incorporating queries as a specialization of functions is a very natural and uniform way of extending the object model to include declarative query capabilities. The major benefits of this approach are:

1. Queries are *first-class* objects, so they support the uniform semantics of ob-

| Native Behaviors | Semantics |
|---|---|
| *B_initialOAPT*(*o*) | Returns the initial Object Algebra Processing Tree (OAPT) resulting from the calculus to algebra translation. This initial OAPT constitutes the initial state of the search space used for the algebraic optimization of the query object *o*. The initial OAPT must be complete for heuristic and randomized search strategies. A heuristic search strategy is used in this thesis to illustrate an optimization scenario for the Query Optimizer (see Appendix A). This behavior is implemented by a stored function. |
| *B_optimizedOAPT*(*o*) | Returns a collection of optimized OAPTs resulting from the optimization process for the query object *o*. This behavior is implemented by a stored function. |
| *B_searchStrat*(*o*) | Returns the search strategy that the optimizer (*B_optimize*) uses to control the optimization of the query object *o*. It must be determined externally, before *B_optimize* is applied (i.e. when the query object *o* is created). |
| *B_transformations*(*o*) | Returns the list of transformation rule objects used for the algebraic optimization of the query object *o*. This behavior is implemented by a stored function. |
| *B_costModelFunc*(*o*) | Returns the cost model function that the optimizer (*B_optimize*) uses when the search strategy (*B_searchStrat*) is a cost-controlled strategy. Otherwise, it returns **null**. It must be determined externally, before *B_optimize* is applied (i.e. when the query object *o* is created). |
| *B_optimize*(*o*) | Executes the algebraic Query Optimizer over the query object *o*, using the search strategy object *o.B_searchStrat* on the initial OAPT object *o.B_initialOAPT* and returns the optimized OAPT *o.B_optimizedOAPT*. This behavior will be invoked by the behavior *B_compile* (which is inherited from `T_function` and redefined in `T_query`). |
| *B_genExecPlan*(*o*) | Generates an execution plan (or a family of execution plans) from the optimized OAPT object *o.B_optimizedOAPT* for the query object *o*. The Execution Plan is modeled as a function object that executes the query object *o*. As a side effect, *B_genExecPlan* stores the resulting Execution Plan so that it can be accessed using the *B_execPlanFamily* behavior. This behavior is invoked by the behavior *B_compile*. |
| *B_execPlanFamily*(*o*) | Returns a collection of execution plans that are generated by *B_genExecPlan*. |
| *B_result*(*o*) | Returns the query result that was stored after the query *o* was executed by applying a particular type of *B_execute* that saves its result (similar to materialization of result relations in relational systems). |

Table 4.2: Behavioral summary of `T_query` type.

jects. They are maintained within the objectbase and are accessible through the behavioral paradigm of the object model.

2. Since queries are objects, they can be used in queries and behaviors can be applied to them. This is useful in generating statistics about the performance of queries and in defining a uniform extensible query optimizer.

3. Queries are uniformly integrated with the operational semantics of the model and thus, queries can be used as implementations of behaviors (i.e., the result of applying a behavior to an object can trigger the execution of a query).

4. The query model is extensible in a uniform way since the type `T_query` can be further specialized by subtyping. This can be used to dichotomize the class of queries into additional subclasses, each with its own unique characteristics, and to incrementally develop the characteristics of new kinds of queries as they are discovered. For example, we can subtype `T_query` into `T_adHoc` and `T_production` and then define different evaluation strategies for each. *Ad hoc* queries may be interpreted without incurring high compile-time optimization strategies while *production* queries may be compiled once and then executed many times.

### 4.2.1 Query Compilation

The behavior *B_compile* that is redefined in `T_query` is responsible for implementing TIGUKAT query methodology that consists of the following steps:

1. Translating the query statement written in TQL language ([27]) into an equivalent calculus expression;

2. Translating the calculus expression into an equivalent algebra expression and checking it for type consistency.

3. Optimizing the algebra expression that is performed by applying the behavior *B_optimize*. Algebraic Optimization consists of applying equivalence preserving rewrite rules to the type consistent algebra expression.

4. Generating an execution plan (or a family of execution plans) which is performed by applying the behavior *B_genExecPlan*. This behavior annotates each individual algebra operator node from the optimized object algebra query processing tree with one of the algorithms that implement the operation represented by the corresponding node. These algorithms use object manager calls that are part of the low level object manipulation primitives that constitute the interface to the object manager subsystem.

The implementation of the first two steps of the methodology is discussed in [27]. This thesis focuses in algebraic optimization (Step (3)) and in the general optimizer architecture. Future research must be done on picking the best algorithms that implement each algebraic node in the OAPT based on information provided by the OM such as indexes, clustering, and so on (Step (4)). Initial work on this step is reported in [43].

As side effects of the application of the behavior *B_compile* on a query object, the values of the following behaviors are set: *B_executable*, *B_initialOAPT*, *B_optimizedOAPT*, *B_transformations*, *B_argMbrTypes*, *B_rcvrMbrTypes*, *B_resultMbrType*, *B_execPlanFamily* and *B_result*.

## 4.3 Query Optimizer as an Object

TIGUKAT query optimizer is incorporated as a behavior *B_optimize* in the interface of the type `T_query`. It is, therefore, modeled as an instance of type `T_behavior` (see Figure 4.1). This means that the query optimizer has the status of *first-class* object in the model.

The query optimizer *B_optimize* is responsible for applying a search strategy *B_searchStrat* to an initial OAPT *B_initialOAPT* in order to produce an "optimal" OAPT *B_optimizedOAPT* for a query object $q$. In case the search strategy is a cost-controlled strategy, the cost model *B_costModelFunc* is used to measure the effects of the optimizer actions. All these behaviors are defined in the interface of the type `T_query` (see Section 4.2).

*B_optimize* is defined on `T_query` and is inherited by its subtypes. The implementation of *B_optimize* and, therefore, of the query optimizer may vary over these types. This gives flexibility in providing different implementations for the optimizer. For example, one implementation may be written in TIGUKAT programming language, while another implementation may use some other object-oriented programming language (i.e. C++). An example of code for the function object that implements *B_optimize* is illustrated in Algorithm 4.1[2]. The current implementation of the query optimizer is discussed in Chapter 8.

**Algorithm 4.1** *Optimizer.:*

*F_optimize*(`T_query q`): `T_list`⟨`T_algOp`⟩
{
    return((`q`.*B_getSearchSS*).*B_execute*(`q`)); **(1)**
}

Before applying the behavior *B_optimize* to a query object $q$, the initial OAPT must have been generated and stored in *B_initialOAPT*; and the query object must have been annotated with the search strategy (*B_searchStrat*) and the cost model function type (*B_costModelFunc*) which are selected externally (i.e. when the query object is created). The cost model is only required when the search strategy is cost-controlled.

Each algebraic operator node has a different cost function because the computation of the cost of executing algebraic operations incorporates optimization issues that potentially vary among the operators. For example, the union operator only needs the cost of accessing the instances of the collections $C_i$, $C_j$ involved in the operation, while the select operator requires the cost of accessing the instances of the collection $C$, in the presence of a predicate $f$. Therefore, the annotation of the OAPT with its algebraic cost function is very important because each node in the OAPT collaborates with the cost model function by calculating its algebraic cost itself. This is further explained in Chapter 7.

[2]In the algorithms that are presented in this thesis, a dot notation $o.b(p)$ is used to illustrate the application of a behavior $b$ to a receiver object $o$ with arguments $p$.

Modeling the building blocks of a cost-based optimizer as objects provides the query optimizer the extensibility inherent in object models. The optimizer basically implements a control strategy that associates a search strategy and a cost function to each query. The database administrator has the option of defining new cost functions and new search strategies or transformation functions for new classes of queries.

# Chapter 5

# Representation of Search Space

The compilation of a query results in the translation of a TQL query to a corresponding algebraic expression represented as an OAPT. The set of equivalent OAPTs that generate the same result make up the search space (i.e., each OAPT corresponds to a state).

While in relational systems, a processing tree is a labelled tree where the leaf nodes represent relations and the intermediate nodes correspond to relational algebra operators, in this research, the nodes in the OAPTs uniformly correspond to functions that model the delayed execution of object algebraic operators.

The modeling as objects of the OAPTs and transformation rules that generate different OAPTs in TIGUKAT Extended Type System is described in Sections 5.1 and 5.2, respectively.

## 5.1 Object Algebra Processing Trees

The object algebra operators are modeled as behaviors on type `T_collection` whose implementations are modeled as instances of `T_function`. Since each algebraic operator has its own characteristics (i.e. predicates, functions to apply, etc), objects of type `T_function` are created according to the different existing operators defined in the TIGUKAT algebra. Furthermore, since there may be a number of different algorithms to implement each algebraic operator (e.g. nested

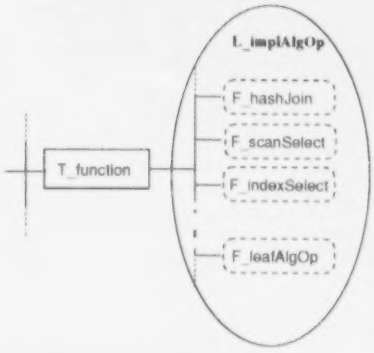


Figure 5.1: Collection (**L_implAlgOp**) of implementations for algebraic operators.

loop join, merge-sort join, and hash join), there may be many implementation functions as instances of `T_function`. For example, *F_scanSelect* is an object of type `T_function` that models the algorithm that implements the select algebraic operation *B_select* defined as a behavior on `T_collection`. This gives flexibility in redefining algebraic operators and in extending the algebra in future versions of the query model to deal with new algebraic operators that may be defined (e.g., transitive closure for recursive query processing). Besides, new algorithms for existent algebraic operators can be added to the system as they are discovered.

In order to clarify the further discussion in the thesis, we define the collection **L_implAlgOp** to group all the instances of `T_function` that implement algorithms for algebraic operators as illustrated in Figure 5.1.

These implementation functions cannot be used as the nodes of an OAPT, however. The nodes of the tree should represent execution functions all of whose arguments have been marshalled (see Figure 5.2). Therefore, we define `T_algOp` whose instances are functions with marshalled arguments and they make up nodes of OAPTs. In this fashion, each node of an OAPT represents a specific execution algorithm for an algebra expression. Instead of defining `T_algOp` as an immediate subtype of `T_function`, we define it as a subtype of `T_context` which, in turn, is a subtype of `T_function`. In a sense, a `T_context` instance corresponds to delayed execution of a function. The reason for the definition of `T_context` and the

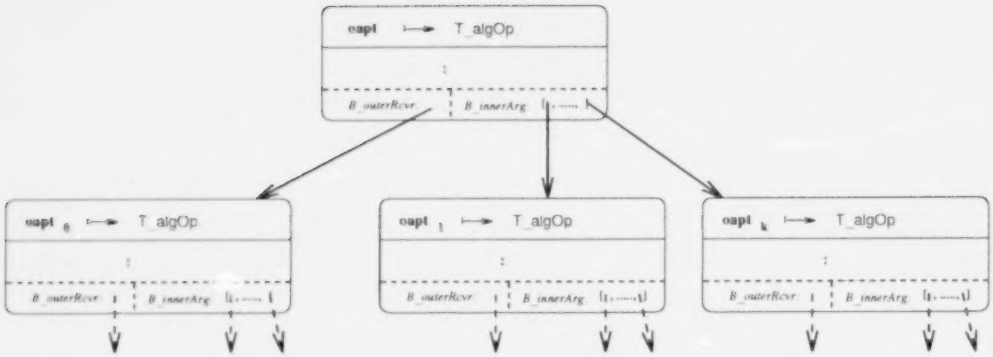


Figure 5.2: Tree shape of an OAPT.

| Native Behaviors | Semantics |
|---|---|
| *B_outerRcvr(o)* | Returns a reference to the context object whose evaluation will serve as the receiver object for the context object *o*. |
| *B_innerArg(o)* | Returns a list of references to the context objects that correspond to the arguments for the behavior whose delayed execution is represented by the context object *o*. |
| *B_rcvrType(o)* | Returns the type of the receiver object for the object *o*. |

Table 5.1: Behavioral summary of `T_context` type.

modeling of `T_algOp` as a subtype of `T_context` is that this allows further optimization possibilities. Since the nodes of OAPTs are instances of `T_context` (due to subtyping), we could relax the restriction that they be instances of `T_algOp` and represent behaviors in predicates of query expressions as nodes in the OAPTs as well. This would open the possibility of optimizing the execution of behaviors together with algebraic operators in a query. Commonly called the *method optimization problem*, this is a serious concern in OBMSs. Even though this research has not addressed this issue yet, the architecture that we has been developed lends itself to such extensions in the future. The native behaviors defined on `T_context` are listed in Table 5.1, and the native behaviors defined on `T_algOp` are listed in Table 5.2.

Since the nodes of OAPTs are instances of type `T_algOp`, to provide uniformity, the object *F_leafAlgOp* is defined to model the leaf nodes of the OAPTs. This object can be thought of as a container that holds a reference to one of the input

| Native Behaviors | Semantics |
|---|---|
| *B_rcvrMbrType*(*o*) | Returns the membership type object of the receiver collection object for the algebraic operator represented by the node object *o*. |
| *B_argMbrTypes*(*o*) | Returns a list whose elements are the membership type objects corresponding to the argument collection objects for the algebraic operator represented by the node object *o*. |
| *B_resultMbrType*(*o*) | Returns the membership type object of the collection produced by executing the algebraic operator represented by the node object *o*. |
| *B_targetVar*(*o*) | Returns a reference to the target variable object for the algebraic operator represented by the node object *o*. |
| *B_targetColl*(*o*) | Returns a reference to the target collection object that results from executing the algebraic operator represented by the node object *o*. |
| *B_constraint*(*o*) | Returns an object that models a constraint on the algebraic operator represented by the node object *o*. For example, a formula that qualifies the select operator, or the list of behaviors that must be applied to the receiver and argument collections of the map operator are constraints on the select and map operators respectively. |
| *B_execAlgorithm*(*o*) | Returns a function object that implements an execution algorithm for the algebraic operation that the node *o* represents. It is implemented by a stored function. |

Table 5.2: Behavioral summary of `T_algOp` type.

collections of the query that the OAPT represents. *F_leafAlgOp* objects model the delayed execution of the algebraic identity operator *B_I* that is defined as part of the interface of the type `T_collection`. Thus, **all** the nodes of an OAPT are uniformly modeled as instances of `T_algOp` rather than making an exception for the leaf nodes which correspond to collections[1].

An OAPT is recursively defined as an object of type `T_algOp` as follows: the root node of the OAPT is an algebraic operator of type `T_algOp` whose children are also of type `T_algOp` with the restrictions that an intermediate node cannot be an object *F_leafAlgOp* and every leaf node is an object *F_leafAlgOp*.

Figure 5.3 shows the OAPT for the algebra expression corresponding to the query that is presented in Example 5.1. The first information in the box represents an object instance reference and the mapping to its type. Then, the behaviors that are relevant to the subsequent discussion are listed. The $f1$ and $f2$ in the figure are the formulas $o.B\_value > 100000 \wedge p.B\_latitude > 10$ and $o.B\_area > 1000$, respectively which are represented as objects of type `T_formula`.

**Example 5.1** Return all the land zones covering an area over 1000 units that are land zones valued over $100,000 and that are located above the latitude 10.

Algebra:

$$(\mathbf{C\_land}_o\ \sigma_{[o.B\_value>100000\ \wedge\ o.B\_origin.B\_latitude>10]})_o\ \sigma_{[o.B\_area>1000]}$$

### 5.1.1 Execution Plan Generation

In relational databases, the execution plan produced by the query optimizer is a processing tree whose algebraic operators can be directly mapped to low level implementation primitives of the physical system (i.e., a tuple can be translated into a record, a table into a file and so on). In contrast, in TIGUKAT object model, algebraic operators are objects that encapsulate their internal representation which can only be manipulated by the Object Manager subsystem. There is

[1]This is a conceptual model; for efficiency reasons, the optimizer may represent the leaf nodes directly as collections and handle them as special cases.

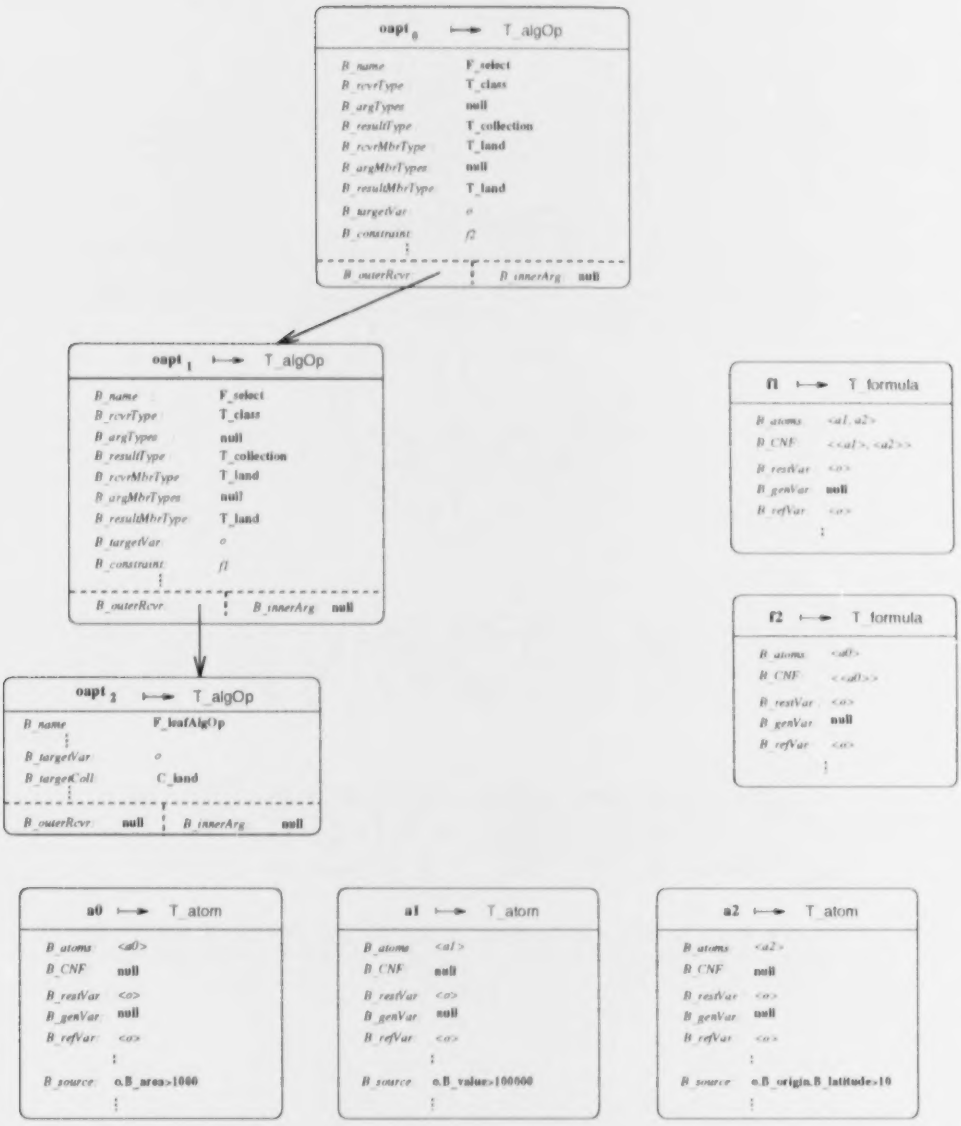


Figure 5.3: Initial OAPT

no one-to-one mapping between the different components of the database and low level primitives of the physical system. Although the execution plan generation is not in the scope of this research, different methodologies that have been proposed to solve this problem are discussed below. It is important to notice that there is no agreement among the database community in the set of low level primitives that an object manager interface should provide.

The execution plan generation method proposed in Straube and Özsu [43] generates the query execution plan by replacing each individual algebra operator from the optimized OAPT with a "best" subtree of Object Manager (OM) calls. These object manager calls are part of the set of low level object manipulation primitives that constitutes the interface to the OM. This approach respects the full encapsulation of objects.

The methodology proposed in the Open OODB project [3] proposes to replace each logical algebra expression by an execution algorithm which is selected by the optimizer based on implementation rules. These rules use physical information that is encapsulated in "physical property vectors" (which is an abstract data type).

This thesis proposes to create a different instance of `T_function` (e.g., nested loop join, merge-sort join, and hash join) for each algorithm that implements an algebraic operator of type `T_algebra` (e.g., join). Then, there may be a number of instances of `T_function`. In this fashion, each node of an OAPT (of type `T_algOp`), that represents an object algebraic operator, is annotated (in the behavior *B_execAlgorithm*) with the function object that implements an specific execution algorithm for that algebra operation. For example, an OAPT node *F_join* is annotated with the function object *F_nestedLoopJoin*. The OAPT node is annotated with default execution algorithms per algebraic operator when the OAPT node is first created.

Algorithm selection can be done by the Execution Plan Generator provided the OM can derive physical information by the application of special behaviors that belong to its Object Manager Interface (OMI). Further research needs to be

done on the OM subsystem and its interface. It is not in the scope of this thesis to define a methodology to select the execution algorithms to annotate the OAPTs with implementation information. This methodology would be implemented by the behavior *B_genExecPlan*. This is topic of future research.

Each node of the OAPT can also be annotated with information about whether the intermediate collection must be materialized or pipelined at execution time. This information is stored in behavior *B_isMaterial* that returns the object **true** if it is materialized or false, otherwise. The behavior *B_targetColl* is a reference to the materialized collection when it is executed.

The Execution Plan is modeled in TIGUKAT Query Optimizer as function objects (`T_algOp` is a subtype of `T_function`). Because a function returns its cost, this approach has the advantage that the cost of the annotated OAPT is the cost of the execution plan for a query.

As discussed in [33], a query execution engine for OBMSs must support the following basic[2] set of algorithms that implement object algebraic operators: *scan*, *indexed scan*, and *collection matching*. These algorithms operate on collections of objects to support the closure of the object algebra on collections. Collection scan is an algorithm that sequentially accesses all objects in a collection. An indexed scan algorithm allows efficient access to objects (that satisfy a selection predicate) in a collection through an index. Collection matching algorithms produce aggregate objects from multiple collections of objects that are given as input to the algorithm. TIGUKAT query optimizer architecture supports the incorporation of these algorithms into the query optimizer by the definition of function objects that implement them such as *F_scanLeaf*, *F_indexSelect*, *F_hashJoin*, and so on. We refer the reader to [33] for a more detailed discussion of query execution since this topic is out of the scope of this thesis.

[2]To support at least the set of algorithms that is provided by relational query execution engines that operate on relations.

### 5.1.2 Execution of the OAPT

As mentioned in Section 5.1, an OAPT is an instance of type `T_algOp` which models the delayed execution of an algebraic expression that represents the query that is being optimized.

After annotating each node of the OAPT with its corresponding implementation, and with information about whether the intermediate collection is materialized or pipelined (*B_isMaterial*), the optimized OAPT can be passed to the Object Manager to be executed as described in Section 4.2 (i.e. by applying the behavior *B_execute* to the query object).

The OAPT is executed bottom-up, where each node consumes operands from left to right. The left child of a node plays the role of receiver of the algebraic operation, and the right child is the list of arguments. The execution ends at the root node. After executing a node, the application of the behavior *B_targetColl* returns a reference to the resulting collection. We do not give more detail about the execution of an OAPT since this is not in the scope of this thesis.

## 5.2 Algebraic Transformation Rules

As indicated before, the search space consists of a family of equivalent plans[3], each of which is represented as an OAPT. The equivalence of two OAPTs corresponding to the same query is established by means of the equivalence-preserving algebraic transformation rules and the semantic rules.

Algebraic transformation rules create equivalent expressions based upon pattern matching and textual substitution. These rules are very dependent upon the specific object algebra because they are defined as combinations of its algebraic operators. In other words, they are specified by algebraic expressions which are functional expressions because of the functional basis of the object algebra ([36]). For example, in [42], the rules are specified using an infix notation, while in [26], they are specified using a prefix notation. An important consideration for defining

[3]These execution plans are equivalent in terms of the results that they generate, but may potentially differ in their costs.

algebraic transformation rules in relational models as well as in object-oriented models is that an algebraic expression representing a query expression can be manipulated using the well-defined algebraic properties such as transitivity, commutativity and distributivity. However, these expressions differ between models in that object algebraic expressions are defined on collections of objects which have an inheritance relationship, while relational algebraic expressions are defined on flat relations. This allows object-oriented query optimizers to use the semantics of the inheritance relationship in order to achieve some additional transformations, called *semantic transformations*. For example, a semantic transformation rule could be defined as follows: if class $C_2$ is a subclass of $C_1$, the intersection of those two classes, $C_1$ and $C_2$, produces the class $C_2$. This results since the object model restricts each object to belong to only one class. This research does not deal with semantic transformation rules and leaves it as a topic of future research.

Algebraic and semantic transformation rules are specified in [42] for a less powerful algebra than the one described in Section 3.2.2; however, no implementation is provided for them. Because of the similar nature of that object algebra and TIGUKAT object algebra, a subset of the algebraic rules specified in [42] are presented in Section 5.2.1 in order to help in the discussion of modeling of the algebraic transformation component as objects and in the discussion of the application of algebraic rules by the query optimizer (that is discussed in Section 5.2.3). The same rules are used in the current implementation of the query optimizer to show the feasibility of our approach. It is not in the scope of this research to specify new transformation rules for those additional algebraic operators, such as join, collapse, reduce, and cartesian product that are introduced by TIGUKAT object algebra. More rules can be added later taking advantage of the extensibility of the transformation rule component of the optimizer.

In order to provide extensibility, algebraic transformation rules need to be modeled as objects in the system. Extensibility is essential in OBMSs where the research into object algebras has not yet matured and new transformation rules may be uncovered as research continues. These transformation rules are

modeled as objects of type `T_algEqRule` that is discussed in Section 5.2.2, which is a subtype of `T_rule`. Although at this stage TIGUKAT does not have "active DBMS" capabilities, we have defined type `T_rule` as an abstract type with subtypes `T_activeRule` and `T_algEqRule`. `T_activeRule` would, in the future, model ECA-type rules [7] when active capabilities are added. The definition of `T_activeRule` is not in the scope of this thesis.

The search strategy determines the use of the rules for controlling the search. There are many alternative strategies, ranging from static priorities to heuristics that determine which rules would be applied under various conditions. This is illustrated in Chapter 6 for an algebraic heuristic search strategy.

### 5.2.1 Specification of Algebraic Transformation Rules

In this section, some of the algebraic equivalence transformation rules that can be applied during optimization to get equivalent query expressions (OAPTs) are presented. They are specified as algebraic expressions using the notation $E_1 \Leftrightarrow E_2$ which specifies that expression $E_1$ is equivalent to expression $E_2$ (e.g., algebraic rule 5.1). Some rules are restricted in that they are applicable only when a condition $c$ is satisfied. They are written as $E_1 \overset{c}{\Leftrightarrow} E_2$ ([9], [42]) (e.g., algebraic rule 5.4). Conditions are conjunction of functions which determine properties of argument collections, predicates and variables used in a rule. These functions are defined as follows: Function $ref(F, (v_1, ..., v_n))$ is true when $v_1, ..., v_n$ are the only variables referenced in the predicate $F$. Function $gen(F, v)$ is true when the predicate $F$ contains a generating atom[4] for the variable $v$. In a similar way, $res(F, v)$ is true when predicate $F$ restricts values of $v$. It can be said that predicate $F$ restricts values of $v$, when it does not contain a generating atom for the variable $v$.

In the algebraic expressions, are introduced as abbreviations for list variables $Q_{list}$, $R_{list}$ and $S_{list}$ to replace $Q_1, ...., Q_k$, $R_1, ...., R_l$, and $S_1, ...., S_m$, respectively, and the symbol '$\Leftrightarrow$' is used to denote equivalence between the left and right side

[4]This is defined in Section 3.2.1.

| Native Behaviors | Semantics |
|---|---|
| *B_cond*(*o*)(*p*) | Returns an object that models the condition that must be satisfied by the object *p* in order to apply the rule *o*. |
| *B_checkCond*(*o*)(*p*) | Checks if the condition stored in *B_cond* holds for the object *p*. If so, the object **true** is returned. Otherwise, **false** is returned. |
| *B_action*(*o*)(*p*) | Returns the object resulting from applying the action dictated by the rule object *o* to the argument object *p*. |

Table 5.3: Behavioral summary of `T_rule` type.

algebraic expressions.

The following examples illustrate possible algebraic rules for TIGUKAT transformation rule component of the optimizer:

**Algebraic Rule 5.1** Commutativity of Select.

$$(P\ \sigma_{f1} < Q_{list} >)\ \sigma_{f2} < R_{list} > \Leftrightarrow (P\ \sigma_{f2} < R_{list} >)\ \sigma_{f1} < Q_{list} >$$

**Algebraic Rule 5.2** Distributivity of Union with respect to Select.

$$(P \cup Q)\ \sigma_f < R_{list} >) \Leftrightarrow (P\ \sigma_f < R_{list} >) \cup (Q\ \sigma_f < R_{list} >)$$

**Algebraic Rule 5.3** Intersection-Select Exchange Rule.

$$(P\ \sigma_{f1} < Q_{list} >)\ \sigma_{f2} < R_{list} >) \Leftrightarrow (P\ \sigma_{f1} < Q_{list} >) \cap (P\ \sigma_{f2} < R_{list} >)$$

**Algebraic Rule 5.4** Conjunctive Select Predicate.

$$(P\ \sigma_{(f1 \wedge f2)} < Q_{list}, R_{list} >) \overset{c}{\Leftrightarrow} (P\ \sigma_{f1} < Q_{list} >) \cap (P\ \sigma_{f2} < R_{list} >) >)$$
$$c\text{: } ref(f1,(p, q_1 ... q_k)) \wedge res(f1, p) \wedge ref(f2,(p, r_1 ... r_k)) \wedge res(f2, p)$$

### 5.2.2 Algebraic Transformation Rules as Objects

Since `T_rule` is a subtype of `T_object`, it inherits all its behaviors and define new ones that are listed in Table 5.3. However, `T_rule` does not implement any of its behaviors because it is an abstract type. These behaviors are implemented by its subtypes.

The algebraic transformation rules are modeled as objects of type `T_algEqRule` that is defined as a subtype of `T_rule`. The native behaviors as well as the redefined inherited behaviors for `T_algEqRule` are described in Table 5.4.

The behavior *B_algExpression* stores the algebraic expression that specifies the rule object. The application of the rest of behaviors is illustrated in Section 5.2.3.

### 5.2.3 Rule Application

The application of algebraic equivalence rules involves checking the *validity* of applying a rule to an specific OAPT and transforming the given OAPT into an equivalent OAPT as specified by the rule.

The process of checking the *validity* of applying a rule is know as *matching* of rules to OAPTs. This process can be done at different levels or steps. In the approach presented here, two levels were identified following the nature of the algebraic rules presented in this research. These two levels are described below,

**Level 1:** Matching the right/left side shape to check if the OAPT matches syntactically with the right/left side of the rule. This is implemented by applying the behaviors *B_matchLeft*/*B_matchRight*

**Level 2:** Testing the condition that checks if the formula satisfies constraints given in the condition of the rule. This matching semantics can be considered computationally more expensive than the level 1 matching. Testing the condition is done only if level 1 matching is satisfied.

After the matching is found to be valid, the transformation is applied to the OAPT in order to get its equivalent OAPT according to that specific rule.

The application of rules by rule based optimizers such as the EXODUS [11] and Starburst [15] optimizers is done by a pattern matching engine that matches subexpressions of a query against algebraic rules. Additionally, the firing of rules is dependent on the satisfaction of the conditions that involve user defined functions such as $res(F, v)$. A major difference between the rules defined for those systems and the ones defined in [42] (which form the basis for specifying TIGUKAT

| Native Behaviors | Semantics |
|---|---|
| *B_leftSideFunc*(*o*)(*p*) | Returns the function object that implements the matching algorithm that corresponds to the left side expression of the rule object *o*. |
| *B_matchLeft*(*o*)(*p*) | Executes the function object stored in *B_leftSideFunc* passing *p* as argument. If the argument object *p* matches the left side of the algebraic equivalence rule, the object **true** is returned. Otherwise **false** is returned. |
| *B_rightSideFunc*(*o*)(*p*) | Returns the function object that implements the matching algorithm that corresponds to the right side expression of the rule object *o*. |
| *B_matchRight*(*o*)(*p*) | Executes the function object stored in *B_rightSideFunc* passing *p* as argument. If the argument object *p* matches the right side of the algebraic equivalence rule, the object **true** is returned. Otherwise **false** is returned. |
| *B_condLeft*(*o*)(*p*) | Returns a function object that implements the condition associated to the left side expression of the rule object *o*. |
| *B_checkCondLeft*(*o*)(*p*) | Executes the function object stored in *B_condLeft*. If the argument object *p* holds the condition associated to the left side of the rule object *o*, the object **true** is returned. Otherwise, **false** is returned. |
| *B_cond*(*o*)(*p*) | Returns a function object that implements for the condition associated to the right side of the rule object *o*. (Inherited from `T_rule`, but overloaded). |
| *B_checkCond*(*o*)(*p*) | Executes the function object stored in *B_cond*. If the argument object *p* holds the condition associated to the right side of the rule object *o*, the object **true** is returned. Otherwise, **false** is returned. (Inherited from `T_rule`, but overloaded). |
| *B_actionLeftFunc*(*o*)(*p*) | Returns the function object that implements the transformation dictated by the left side of the rule object *o* to the argument object *p*. |
| *B_actionLeft*(*o*)(*p*) | Executes the function object stored in *B_leftSideFunc* passing *p* as argument. It returns the OAPT object resulting from applying the transformation dictated by the rule object *o* to the argument object *p*. The resulting OAPT have the shape of the algebraic expression for the right side of the rule object *o*. |
| *B_actionRightFunc*(*o*)(*p*) | Returns the function object that implements the transformation dictated by the right side of the rule object *o* to the argument object *p*. |
| *B_action*(*o*)(*p*) | Executes the function object stored in *B_actionRightFunc* passing *p* as argument. It returns the OAPT object resulting from applying the transformation dictated by the rule object *o* to the argument object *p*. The resulting OAPT have the shape of the expression given in the left side of the rule. (Inherited from `T_rule`, but overloaded). |
| *B_algExpression*(*o*) | Returns the algebraic expression that specifies the rule object *o*. |

Table 5.4: Behavioral summary of `T_algEqRule` type.

algebraic rules) is that the rules for the former systems are based on operators of fixed arity (i.e. two operand joins), while the rules for the latter ( and for TIGUKAT) are based on algebraic operators which can have varying numbers of arguments. As pointed out in [42], in order to use those rule based optimizers for the application of the rules defined herein, their pattern matching engine would have had to be modified to handle algebraic operators which can have varying numbers of arguments. Considering that TIGUKAT query optimizer is an extension of TIGUKAT object model which differs from those systems, we believe that there would be problems integrating those systems with the TIGUKAT query optimizer. Therefore, a pattern matching engine for TIGUKAT algebraic rule component was built.

The proposed definition of OAPT nodes as objects of type `T_algOp`, and the closure of the algebra on collections enables these operators to be used as a recursive functional symbol for describing OAPTs in a syntactical way. For patterns that correspond to the algebraic expressions given in an algebraic equivalence rule, a tree representation is convenient. At the same time that the matching between the OAPT and the corresponding pattern for a rule is checked, the binding between them is stored. In this approach, additional structures are needed to keep the binding between the pattern and the OAPT. One disadvantage is that in order to allow concurrency in the optimization of queries in a multi-user environment, the optimizer may have to create several temporary objects to establish the bindings between the pattern trees and the OAPT to which the rule is trying to be applied.

A completely different approach is that each rule provides individualized behavior for its matching and transformation steps. For each rule, there are function objects that implement those steps that may be potentially different among the rules. The pattern matching may be implemented by several function objects, instead of having one general pattern matching algorithm and special structures to represent the pattern as done in the approach presented above. Therefore, the application (matching/transformation) of a rule is done in an object-oriented fash-

ion as described below. In order to help to clarify this discussion, the application of an specific rule (Rule 5.1) is used.

**Matching:** For a rule object *o*, matching at the first level is performed by the application of behaviors *B_matchLeft*/*B_matchRight* that execute the function object returned by *B_leftSide*/*B_rightSide*, respectively, depending on what side of the rule is applied. For example, if the left side of the rule is to be applied, the Algorithm 5.2 that implements *B_matchLeft* executes the function object returned by *B_leftSide* (line 1) whose source code is illustrated in Algorithm 5.1.

**Algorithm 5.1** *Source code for function returned by B_leftSide for select-exchange rule:*

```
F_leftSide_selExch(T_algEqRule o, T_algOp p): T_boolean
{
    if (p.B_name = 'F_select') then                          (1)
        if (p.B_outerRcvr.B_name = 'F_select') then          (2)
            return (true)                                    (3)
        else                                                 (4)
            return (false)                                   (5)
    else                                                     (6)
        return (false)                                       (7)
}
```

**Algorithm 5.2** *Source code for function that implements B_matchLeft for select-exchange rule:*

```
F_matchLeft_selExch(T_algEqRule o, T_algOp p): T_boolean
  T_boolean match;

{
    match = B_leftSide.B_execute(p);                         (1)
    return (match);                                          (2)
}
```

| Behaviors | Semantics |
|---|---|
| *B_atoms(o)* | Returns the list of atoms that are referenced in the formula object *o*. It is implemented by a stored function. |
| *B_CNF(o)* | Returns a representation for the formula, which is given in Conjunctive Normal Form. |
| *B_restVar(o)* | Returns the list of restricted variables in the atoms that are referenced by the formula object *o*. Restricted variables are variables that are not generated by any atom in the formula *o*. It is implemented by a computed function. |
| *B_genVar(o)* | Returns the variable that is generated by one of more atoms in the formula object *o*. A formula can have only one generated variable because of constraints in the object algebra. It is implemented by a computed function. |
| *B_refVar(o)* | Returns the list of variables that are referenced in the atoms of the formula *o*. It is implemented by a computed function that performs the union between the list *B_restVar* and the list containing the generated variable *B_genVar*. |
| *B_splitRestrDisj(o)(p)* | Returns the collection of formulas in CNF form that result from splitting the formula *o* into *p* number of parts which were connected by disjunctions. |
| *B_splitRestrConj(o)(p)* | Returns the collection of formulas in CNF form that result from splitting the formula *o* into *p* number of parts which were connected by conjunctions. |

Table 5.5: Behavioral summary of `T_formula` type.

**Algorithm 5.3** *Source code for function returned by B_condLeft for conjunctive select-predicate rule:*

```
F_condIs_conj_sel(T_algEqRule o, T_algOp p): T_boolean
{
    f = p.B_constraint;   % extracts formula from OAPT          (1)
    CNF= f.B_splitRestDisj;                                     (2)
    if (CNF ≠ null) then                                        (3)
            return (true)                                       (4)
        else                                                    (5)
            return (false)                                      (6)
}
```

Matching at the second level is performed by the application of behaviors *B_checkCondLeft*/*B_checkCond* that execute the function object returned by *B_condLeft*/*B_cond*, respectively. The source code for these function

| Native Behaviors | Semantics |
|---|---|
| *B_splitLeft*(*o*) | Returns the subtree that corresponds to the *B_outerRcvr* of the OAPT *o*. As a side effect, it sets to **null** the behavior *B_outerRcvr* for the node *o*. |
| *B_splitRight*(*o*) | Returns the list of OAPTs that corresponds to the *B_innerArgs* of the OAPT *o*. As a side effect, it sets to **null** the behavior *B_innerArgs* for the node *o*. |
| *B_linkLeft*(*o*)(*p*) | Links the OAPT *p* to the node *o* as *o*'s left son *B_outerRcvr*. Returns the node *o*. |
| *B_linkRight*(*o*)(*p*) | Links the list of OAPTs *p* to the node *o* as *o*'s right son *B_innerArgs*. Returns the node *o*. |
| *B_assemble*(*o*)(*p*)(*q*) | Given the OAPT *p*, the list of OAPTs *q* and the node *o*, *B_assemble* combines them into a single OAPT with root *o*, left son (*B_outerRcvr*) *p*, and right son (*B_innerArgs*) *q*. Returns the OAPT rooted at *o*. |
| *B_dissassemble*(*o*) | Breaks the OAPT rooted at *o* into three parts: an OAPT containing only the node *o*, and the left and right children of *o*. Returns a list containing at most two elements: the first element is the left son (*B_outerRcvr*), and the second element is the right son (*B_innerArgs*). As a side effect, it sets to **null** the behaviors *B_outerRcvr* and *B_innerArgs* for the OAPT *o*. |

Table 5.6: Behavioral summary of operations on trees defined on `T_algOp` type.

objects is expressed in terms of operations that manipulate formulas to determine properties of predicates and variables used in a rule (i.e. $gen(F, v)$). In TIGUKAT, these operations are modeled as behaviors defined in the interface of `T_formula` as illustrated in Table 5.5. For example, for Rule 5.4, Algorithm 5.3 illustrates the source code for the function object stored in *B_condLeft*. This behavior is set to **null** for Rule 5.1 because the specification for this rule does not establish any restriction.

**Transformation:** The transformation (action) is performed by the application of behaviors *B_actionLeft*/*B_action* that execute the function object returned by *B_actionLeftFunc*/*B_actionRightFunc*, respectively. The source code for these function objects is expressed in terms of operations on trees[5] that in TIGUKAT are modeled as behaviors defined in the interface of `T_algOp` as illustrated in Table 5.6. For example, Algorithms 5.4 and 5.5 illustrates

[5]The operations on trees refered in this research are adapted from the ones presented in Tarjan [40].

the source code corresponding to the function object that is returned by the application of the behavior *B_actionLeftFunc* for Rules 5.1 and 5.4, respectively.

**Algorithm 5.4** *Source code for function returned by B_actionLeftFunc for select-exchange rule:*

```
F_actionLeft_selExch(T_algEqRule o, T_algOp r): T_algOp
  T_algOp sl1, sl2;

{
    sl1 = r.B_splitLeft();                    (1)
    sl2 = sl1.B_splitLeft();                  (2)
    r = r.B_linkLeft(sl2);                    (3)
    sl1 = sl1.B_linkLeft(r);                  (4)
    return(sl1);                              (5)
}
```

As a result of this discussion, for a rule object *o*, the algebraic expression (*B_algExpression*) that specifies the rule is a theorem. The function code that implements the matching (*B_leftSide* / *B_rightSide*), the check condition (*B_condLeft* / *B_cond*), and the transformation (*B_actionLeftFunc* / *B_actionRightFunc*) together are the proof of the theorem.

The search strategy determines the use of the rules for controlling the search. There are many alternative strategies, ranging from static priorities to heuristics that determine which rules would be applied under various conditions.

The example shown in Figure 5.4 illustrates the resulting OAPT after applying the Algebraic Rule 5.1 to the OAPT shown in Figure 5.3.

**Algorithm 5.5** *Source code for function returned by B_actionLeftFunc for conjunctive select-predicate rule:*

```
F_actionLeft_conj_sel(T_algEqRule o, T_algOp oapt): T_algOp
  T_algOp sl, oapt_copy, oapt_root;
  T_list sr; innerArgs
  T_formula formula;

{
    sl = oapt.B_splitLeft();                                              (1)
    sr = oapt.B_splitRight();                                             (2)
    oapt_copy = oapt.B_copyComp();                                        (3)
    formula = oapt.B_setFormula(o.B_CNF.B_first.B_first);                 (4)
    formula = oapt_copy.B_setFormula(o.B_CNF.B_first.B_last);             (5)
    oapt = oapt.B_linkLeft(sl);                                           (6)
    oapt_copy = oapt_copy.B_linkLeft(sl);                                 (7)
    % sr_1 = select target collections ref by var in formula F1          (8)
    % sr_2 = select target collections ref by var in formula F2          (9)
    oapt = oapt.B_linkRight(sr_1);                                        (10)
    oapt_copy = oapt_copy.B_linkRight(sr_2);                              (11)
    innerArgs = C_list.B_new();                                           (12)
    innerArgs.B_setFirst(oapt_copy);                                      (13)
    oapt_root = C_algOp.B_new(F_union);                                   (14)
    oapt_root = oapt_root.B_setTargetVar(oapt.B_getTargetVar);            (15)
    oapt_root = oapt_root.B_assemble(oapt,innerArgs);                     (16)
    return(oapt_root);                                                    (17)
}
```

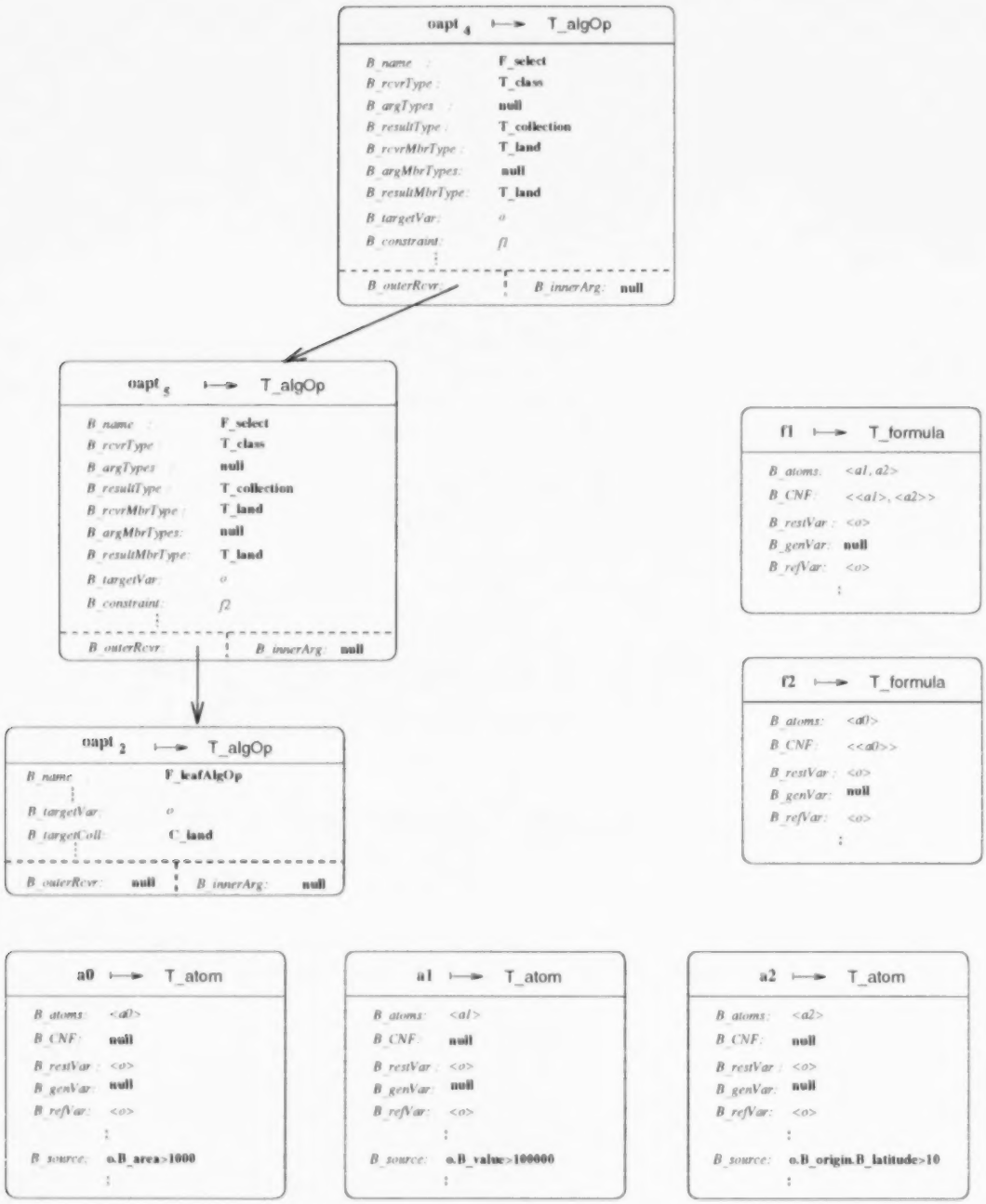


Figure 5.4: Transformed OAPT by using Rule 5.1

# Chapter 6

# Modeling of Search Strategies

In traditional optimizers, the search space (usually identified as a set of processing trees) and the search strategies that control the movement through this search space are coupled together. However, in an extensible query optimizer, they need to be decoupled. Consequently, the type `T_searchStrat` is defined to model search strategies as objects. This type is a subtype of the type `T_function` in the extended TIGUKAT type system.

`T_searchStrat` is an abstract type whose behaviors are implemented in its subtypes. In other words, its extent is empty. This type is specialized into heuristics-based optimization strategies `T_heurSS`, enumerated search strategies `T_enumSS`, and randomized search strategies `T_randomSS` (as shown in Figure 4.1). Then, search strategies are modeled as first-class objects whose type is any of the types in the `T_searchStrat` type hierarchy.

The search strategy component of the query optimizer can be easily extended by further subtyping the type hierarchy for `T_searchStrat` when new search strategies are found useful for the optimization of queries.

Before optimizing a query, the system selects the appropriate search strategy for the particular type of query under consideration. One criterium involved in this selection can be to achieve a desirable trade-off between optimization cost and execution cost for that query. For example, for ad-hoc queries (of type `T_adHoc`), a heuristic search strategy may be desired since high cost of optimization may not be

amortized over repeated executions of the query. In contrast, production queries (of type `T_production`), may be optimized more elaborately by using strategies such as exhaustive search since the optimization is done at compile time and the resulting optimized execution plan is stored and potentially executed many times.

The flexibility of supporting various search strategies, each one best for a particular class of queries, and the dichotomization of classes of queries by subtyping `T_query` enable to associate a default search strategy to each type in the `T_query` hierarchy that suits better the optimization requirements for that class of queries.

The behavior *B_searchStrat* that is incorporated into the interface of the type `T_query` returns the search strategy object that was chosen to be used to optimize that query[1] before the optimization starts. This search strategy can be executed to control the application of transformation rules to the states (OAPTs) in the search space during query optimization. This allows experimentation with different search strategies for optimizing the same query by plugging in a different search strategy object to the behavior *B_searchStrat* in order to identify the search strategies that are more suitable for a class of queries.

## 6.1 Search Strategies

### 6.1.1 Algebraic Heuristic Search Strategies

An algebraic heuristic search strategy transforms an initial OAPT into an optimized OAPT by applying algebraic equivalence transformation rules (as those defined in Section 5.2). The application of the transformation rules is guided by heuristic query optimization rules.

In the relational model, there are a number of transformations that have been found to be useful in optimizing an algebraic expression. For example, pushing a select operation past a join operation is generally accepted to be a useful transformation. This heuristic assumes that the join operation is an expensive operation,

[1] When instances of a type in the `T_query` type hierarchy are created, a search strategy may be associated by default.

while the selection operation is straightforward or could even be done as part of the joining process.

The idea behind these heuristics ([48]) is to try to reduce as much as possible the size of relations that are the operands of expensive operations such as join, union and intersection. This reduction in size is achieved by pushing unary operations such as selection (to reduce the number of tuples) and project (to reduce the number of attributes in the relation) past the expensive operations as far down the processing tree as possible. In addition, operations that generally reduce the number of tuples in the result, such as join and select, should be executed before other binary operations that are considered expensive such as union, and intersection. In order to achieve this, reordering of the leaf nodes of the tree may be necessary (i.e. commutativity rules).

It is important to notice that in heuristic search strategies only one processing tree is generated after a transformation has been applied by the optimizer. Other search strategies, such as a breadth-first enumerative search strategy, generate a set of candidate trees to be explored by the search algorithm.

In this research, a heuristic search strategy applied to object-oriented queries is illustrated for TIGUKAT query model. The heuristics that are used to guide the search are modeled by a list of transformation rules that is ordered by importance of applicability of the rule (priority given to the rule). This is discussed in more detail in Section 6.3.

### 6.1.2 Cost-controlled Heuristic Search Strategies

In order to determine the applicability of these heuristics in object-oriented query models, it has been found ([30, 26]) that in the presence of methods (behaviors in the case of TIGUKAT) in a selection predicate of a query, it becomes important to determine the cost of applying those behaviors.

In order to solve this problem for object-oriented query models, a cost-controlled heuristic search strategy that considers the cost of applying algebraic equivalence rules has been suggested in [25]. In order to respect encapsulation, this requires

cost models for high-level expressions. These cost models can be expressed in terms of costs derived for calls to an OM interface as suggested in [43]. This is discussed in more detail in Chapter 7.

### 6.1.3 Enumerative Search Strategies

Exhaustive search, whereby the entire search space is enumerated, is the most straight-forward search strategy that can be used. Then, the cost function can be applied to all of these equivalent expressions to determine the least costly one. However, the computational cost of this approach is very high. An improvement is to use a dynamic programming approach whereby new expressions are constructed bottom-up using the previously determined optimal subexpressions. The Volcano optimizer generator uses a top-down, dynamic programming approach to search with branch-and-bound pruning [13]. These are called *enumerative algorithms*.

Enumerative search algorithms are based on evaluating the cost of the entire space.

### 6.1.4 Randomized Search Strategies

Randomized search algorithms start from a random state in the search space (i.e. the initial OAPT that was obtained from the calculus to algebra translation) and then "walk" through the search space, evaluating the cost of each state and stopping either when they estimate that they have found the optimum execution plan or when a predetermined optimization time expires. The walking between states is controlled by the transformation rules such as the ones described in Section 5.2 and a global control strategy. Two versions of these algorithms have been investigated within the context of relational query optimization: *Simulated Annealing (SA)* [17] and *Iterative Improvement (II)* [46, 45]. A combination of the two algorithms, called *two-phase optimization* is proposed in [16]. Iterative improvement accepts a move from one state to another only if the cost of the destination state is lower than the cost of the source state. Simulated annealing, on the other hand, allows a move to a higher-cost state with a certain probability

| Native Behaviors | Semantics |
|---|---|
| *B_initSS(o)* | Returns the initial OAPT(s) of the search space from where the search strategy object *o* starts the search. As a side effect, it may initialize other variables that are relevant to the particular search strategy. |
| *B_stopCond(o)* | Returns the object **true** if the condition given to stop the search process *o* holds. Otherwise **false** is returned. This is useful in modeling randomized search strategies. |
| *B_setNextState(o)(p)* | Returns the next state after *p* in the search space. It determines the order that the states are investigated in the search space. For example, in enumerative search strategies, if the implementation of this behavior chooses the least recent state, then the search strategy is breadth-first; if it chooses the most recently generated state, then it implements depth-first search. |
| *B_action(o)(p)* | Generates a list of successor states for the state *p* by applying algebraic equivalence transformation rules on it. |
| *B_goal(o)* | Returns the collection (set) of states that have been chosen as "good" candidates to be the optimized OAPT. |
| *B_optimal(o)* | Returns the "optimal" OAPT from the *B_goal* collection. |

Table 6.1: Behavioral summary of `T_searchStrat` type.

which diminishes as optimization time moves along.

Randomized search algorithms have been suggested as one alternative to restrict the region of the search space that is analyzed. Since these are heuristic algorithms investigating only a portion of the search space, they cannot be guaranteed to be optimal. However, it has been shown in [46, 45, 17, 16] that the randomized techniques converge to a state which is fairly close to the optimal state given sufficient time.

There has not been any study of randomized search algorithms within the context of OBMSs. However, the architecture defined in this thesis is sufficiently flexible that such strategies can easily be incorporated.

## 6.2 Search Strategies as Objects

Since `T_searchStrat` is a subtype of `T_function`, it inherits all its behaviors (which are described in Section 4.2) and define new ones that are listed in Table 6.1. However, `T_searchStrat` does not implement any of its behaviors because

it is an abstract type. These behaviors are implemented by its subtypes.

The search algorithm is modeled as behaviors *B_source* (source code) and *B_executable* (executable code). These are part of the interface of a type in the `T_searchStrat` type hierarchy. This algorithm is expressed in terms of other behaviors that are part of the interface of the same type as well. This means that a search strategy object can be considered as a program whose source code is stored in *B_source*, its local variables are modeled as stored behaviors in the interface of its type (i.e. *B_current* defined in the interface of `T_heurSS`), and its actions are modeled as behavioral applications of behaviors that are defined in its interface as well (i.e. *B_action* defined in the interface of `T_heurSS`). Furthermore, the comments that document the program are stored in *B_comments* (which is inherited from `T_function`). Therefore, in this research, programs are modeled as objects which is an approach that is used in object-oriented programming languages, but not in databases.

As it was mentioned earlier in this chapter, before the optimizer *B_optimize* is applied to a query object `q` (`q`.*B_optimize*()), the query object `q` must be annotated with an instance of the appropriated search strategy type. For example, if an algebraic heuristic search strategy is desired to be used to control the optimization of a particular query, an instance of type `T_heurSS` must be created and plugged in the behavior *B_searchStrat* for the query object `q`.

## 6.3 Customizing the Search Strategy

The customization of the search strategy component is done by subtyping `T_searchStrat` (i.e. into `T_heurSS`) and overloading its behaviors (i.e. *B_setNextState*) which are called *extensibility behaviors*.

The extensibility behaviors capture common aspects of various known search strategies such as heuristics, enumerative and randomized ones. In order to illustrate this, a customization for an algebraic heuristic search strategy is shown in this section.

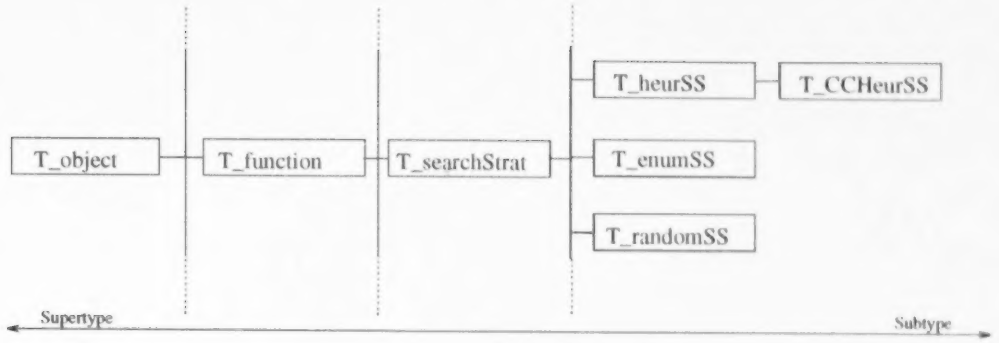


Figure 6.1: Type hierarchy for `T_searchStrat`.

**Algorithm 6.1** *Algebraic Heuristic Search Algorithm.:*

*F_heurSS* (`T_heurSS` *ss*, `T_query` *q*) : `T_list`⟨`T_algOp`⟩

**Input:** The search strategy object *ss* that is implemented by this function
A query object to be optimized *q*

**Output:** A list with only one element: the "optimal" OAPT $oapt_f$

```
T_algOp oapt, oapt1, oapt_f;
T_list oaptList;

{
    oapt = ss.B_initSS(q);                                  (1)
    while not ss.B_stopCond()                               (2)
    {                                                       (3)
        oaptList = ss.B_action(ss.B_getCurrent());          (4)
        oapt1 = ss.B_setNextState(oaptList);                (5)
        if ss.B_acceptAction(oapt1) then                    (6)
            oapt1 = ss.B_setCurrent(oapt1);                 (7)
    }                                                       (8)
    oapt_f = ss.B_setOptimal(ss.B_getCurrent());            (9)
    return(oapt_f);                                         (10)
}
```

`T_searchStrat` is subtyped into `T_heurSS` (see Figure 6.1) which is the type for a heuristic search strategy. Since `T_heurSS` is a subtype of `T_searchStrat`,

| Native Behaviors | Semantics |
|---|---|
| *B_current(o)* | Returns the current state of the search space that is being explored by the heuristic search strategy *o*. |
| *B_acceptAction(o)(p)* | Returns **true** if the OAPT *p* meets the criteria that is defined for the heuristic search strategy. (i.e. if the OAPT is a bushy or a linear tree). Otherwise, it returns **false**. |
| *B_transfRules(o)* | Returns the list of transformation rules that are applied by the search strategy *o*. It must be set when the search strategy object *o* is first created. The list is ordered by priority of the rule. |
| *B_chooseRule(o)* | Returns the current element in the list *B_transfRules*. The next element becomes the current one. |

Table 6.2: Behavioral summary of `T_heurSS` type.

it inherits all its behaviors and define new ones. For a heuristic search strategy, *B_execute* executes its search algorithm that is stored in *B_source* (see Algorithm 6.1). The native behaviors that are defined on `T_heurSS` are listed in Table 6.2 and the customization of the extensibility behaviors is illustrated in Table 6.3.

The behavior *B_initSS* initializes the current OAPT *B_current* with the initial OAPT that results from the calculus to algebra translation. The application of the heuristics that guide the application of transformation rules is controlled by the behavior *B_action*. The rules are applied in the same order to each node in the OAPT. The order is determined by *B_chooseRule* in collaboration with *B_action*. After *B_action* applies a transformation rule to the current OAPT (*B_current*), it returns a list that contains only one element – the transformed OAPT. This list contains only one element because of the particular characteristics of a heuristic search strategy. Then, *B_setNextState* simply returns the first and only OAPT in the list. Next, the search algorithm checks whether the transformation can be accepted or not by applying the behavior *B_acceptAction* to the transformed OAPT. For example, a criteria to determine if a transformed OAPT can be accepted is that one that restricts the search space to contain OAPTs of a particular shape (i.e. bushy or linear trees). This process is repeated until all the rules have been applied to all the nodes of the OAPT (*B_stopCond*).

| Extensibility behaviors | Heuristic | Cost-Controlled Heuristic |
|---|---|---|
| *ss.B_initSS(q)* | `T_algOp` *oapt*;<br>*oapt* = *ss.B_setCurrent*();<br>(*q.B_getInitialOAPT*());<br>return *oapt*; | `T_algOp` *oapt*;<br>*oapt* = *ss.B_setCurrent*().<br>(*q.B_getInitialOAPT*());<br>*cf* = (*q.B_getCostModelFunc*());<br>*cost* = *cf.B_execute*<br>(*ss.B_getCurrent*());<br>*cost* = *ss.B_setCurrCost*(*cost*);<br>return *oapt*; |
| *ss.B_stopCond*() | return *ss.*<br>*B_getTransfRules*().<br>*B_isEmpty*()<br>% No more rules<br>% to apply | return *ss.B_getTransfRules*().<br>*B_isEmpty*()<br>% No more rules to apply |
| *ss.B_action(o)* | `T_algEqRule` *rule*;<br>*rule* = *ss.B_chooseRule*().<br>(*ss.B_getTransfRules*());<br>return *o.B_transform*(*rule*); | `T_algEqRule` *rule*;<br>*rule* = *ss.B_chooseRule*().<br>(*ss.B_getTransfRules*());<br>return *o.B_transform*(*rule*); |
| *ss.B_setNextState(o)* | return *o.B_getFirst*(); | return *o.B_getFirst*(); |
| *ss.B_acceptAction(o, q)* | if (*o* satisfies criteria)<br>return true<br>else<br>return false | `T_integer` *cost1*;<br>*cost1* = (*q.B_getCostModelFunc*()).<br>*B_execute*(o);<br>if (*o* satisfies criteria)<br>if (*cost1* < *ss.B_getCurrCost*())<br>{<br>*cost1*=*ss.B_setCurrCost*(*cost1*);<br>return true<br>}<br>else<br>return false;<br>else<br>return false; |

Table 6.3: Extensibility behaviors for implementing heuristic and cost-controlled heuristic search strategies.

| Native Behaviors | Semantics |
|---|---|
| *B_currCost(o)* | Returns the cost of the current state of the search space that is being explored by the cost-controlled heuristic search strategy *o*. |

Table 6.4: Behavioral summary of T_CCHeurSS type.

## 6.4 Extending the Search Strategy Component

In order to illustrate the extensibility of the search strategy component of the optimizer, a cost-controlled heuristic search strategy is incorporated into the system that is modeled by the type T_CCHeurSS. The search algorithm for this new search strategy is the same as the one given for the non-cost-controlled heuristic search strategy (see Algorithm 6.1). However, some of its extensibility behaviors such as *B_initSS* and *B_acceptAction* are redefined in T_CCHeurSS. For this reason, T_CCHeurSS is created as a subtype of T_heurSS (see Figure 6.1).

The customization of the extensibility behaviors for the cost-controlled heuristic search strategy is illustrated in Table 6.3, and the native behaviors that are defined on T_CCheurSS are listed in Table 6.4.

The cost-controlled heuristic search strategy is customized from the algebraic heuristic search strategy (T_heurSS) by incorporating into the semantics of *B_acceptAction* additional criteria to decide whether or not the transformation applied to the OAPT *B_getCurrent* is accepted. The goal of this additional criteria is to keep the OAPT with the lowest cost as the current OAPT. Then, the cost of the transformed OAPT *oapt1* is computed and compared to the cost of the current OAPT *B_getCurrent* in order to make the decision. A cost function *B_costFunction*[2] is used in order to compute the cost of an OAPT. The application of the behavior *B_costFunction* to an OAPT is explained in Chapter 7. The behavior *B_cost* executes the function returned by *B_costFunction* (*o.B_costFunction*().*B_execute*()).

[2] *B_costFunction* is inherited from T_function, but it is re-defined in the interface of T_algOp

# Chapter 7

# Modeling of Cost Functions

The final optimization-related concept that needs to be incorporated into the model is the cost function. Cost-based optimization strategies apply a predetermined cost function (total time or response time) to an OAPT to calculate the cost of executing the corresponding query according to that OAPT. The issue is how these cost functions are modeled.

In TIGUKAT, each function is associated a cost through *B_costFunction*. Application of this behavior to a function $f$ returns a function object $o$ of type `T_costFunc` that implements the computation of the cost of executing the function $f$. When function $o$ is executed, it returns the actual cost of executing function $f$. For this reason, `T_function` is further subtyped into `T_costFunc`.

Algebraic operator context nodes (instances of `T_algOp`) redefine the behavior *B_costFunction* to return a function object of type `T_costFuncAlgOp`, which is a subtype of `T_costFunc`. This redefinition is necessary since the cost of executing algebraic operators requires the incorporation of various optimization issues into these functions. These issues are typical ones such as the availability of indexes over the collection on which these operators are defined, the statistical information about these collections, and so on. The function object returned by *B_costFunction* when it is applied to an OAPT node is called *algebraic node cost function* for the remainder of this thesis and is denoted as *anc*.

For an algebraic operator context node $o$, the cost of executing the node $o$ in-

cludes the cost of executing the algebraic operator corresponding to that context node *o* plus the cost of executing the children of node *o*. Therefore, this computation is recursive in nature. The cost of a node is determined in terms of the cost of its children whose cost in turn depend on the cost of their own children. Recursion naturally stops at the leaf nodes. Thus, this definition of cost functions is based on graphs. Given an OAPT, its total time is calculated by summing up the costs of all of its nodes; the calculation of the response time is dependent upon the shape of the OAPT (i.e., bushy trees vs. linear trees) and whether parallel execution is possible (these cost functions are described in Section 7.1).

Because of the nature of these cost functions, they are modeled as function objects (of type `T_costFunc`) in TIGUKAT that are passed the root of an OAPT (or subtree) as parameter. Consequently, the application of one of these function objects to the root context of an OAPT calculates the estimated cost of executing that query according to the execution plan represented by that OAPT. We refer to the cost of executing an OAPT as the *cost model function cf* for the rest of this chapter.

A cost model function *cf* is selected for each query[1] before the optimization starts, and it is stored as the value of the behavior *B_costModelFunc* that is incorporated into the interface of the type `T_query`. Then, different cost model functions can be used at different times for optimizing a particular query or type of queries under consideration. This allows later experimentation by changing the cost function in order to measure the effects of the optimizer actions on the optimization of a particular query or type of queries.

In order to clarify the discussion, we refer to the cost model function illustrated in Section 7.1.1, that calculates the total time of execution of a query, for the remainder of the chapter. The concepts developed in this chapter apply to other cost models (i.e. to compute response time that is described in Section 7.1.2). Then, the modeling of cost model functions as objects is discussed in Section 7.2.

[1] When instances of a type in the `T_query` type hierarchy are created, a cost model may be associated by default.

## 7.1 Cost Model Functions

Cost functions such as total time and response time are traditionally used in databases to calculate the estimated cost of executing a query according to the execution plan represented by an equivalent processing tree. We adapt these two cost functions to calculate the estimated total time and response time of executing OAPTs (TIGUKAT's processing trees) as described below in Sections 7.1.1 and 7.1.2, respectively.

### 7.1.1 Total Time Cost Function

A cost model function *tt* that calculates the total time of execution for a particular OAPT rooted at node *oapt* is defined by the equation[2] given below,

$$tt(oapt) = \begin{cases} 0 & \text{if } oapt \text{ is a leaf node} \\ anc(oapt) + tt(oapt.B\_outerRcvr) & \\ + \sum_{i=1}^{k} tt(oapt.B\_innerArgs_i) & \text{otherwise} \end{cases} \tag{7.1}$$

where *anc* is the algebraic node cost function, the left son of *oapt* is the receiver node (*B_outerRcvr*) and the right son is the list of arguments (*B_innerArgs*) which has $k$ elements as illustrated in Figure 5.2. These two behaviors are defined in the interface of `T_context` (see Table 5.1).

The function *tt* is defined recursively. It adds the estimated algebraic cost of executing a node *oapt* (by applying *anc*) to the cost of executing *oapt*'s children which is computed by applying the same function *tt* to each of them. The total time is calculated by summing up the costs of all nodes in the OAPT. The cost of executing leaf nodes is considered negligible ($anc(leaf) = 0$) because leaf nodes are references to base collections. Then, $tt(leaf) = 0$. Recursion stops at the leaf nodes.

The algebraic node cost function *anc* may involve the cost of accessing the objects in secondary storage, the cost of processing the algebraic operation and

[2] A cost model is described by an equation that is implemented by a function object in TIGUKAT extended model.

the cost of storing the intermediate results. In the case of a distributed database, it may also involve the cost of communication between the node and its children as considered in [8]. A discussion of algebraic node cost functions is given in Section 7.2.1.

### 7.1.2 Response Time Cost Function

A cost model function *rt* that calculates the response time of execution for a particular OAPT rooted at node *oapt* is defined by the equation given below,

$$rt(oapt) = \begin{cases} 0 & \text{if } oapt \text{ is a leaf node} \\ anc(oapt) & \\ +max(rt(oapt.B\_outerRcvr), & \\ \quad max_{i=1}^{k}(rt(oapt.B\_innerArgs_i))) & \text{otherwise} \end{cases} \tag{7.2}$$

where *anc*, *B_outerRcvr*, and *B_innerArgs* have the same definition given in Section 7.1.1.

The function *rt* is defined recursively as well. It finds the cost of the most costly path in the OAPT by traversing the tree from root to leaves and computing the cost of each node in terms of the response time cost of its children. The cost of executing leaf nodes is considered negligible ($anc(leaf) = 0$) because leaf nodes are references to base collections. Then, $rt(leaf) = 0$. Recursion stops at the leaf nodes.

## 7.2 Cost Model Functions as Objects

Cost model functions are modeled as recursive functions that take the root of an OAPT as input (parameter), and return the cost of the OAPT as result, after the function has recursively traversed the OAPT calculating the cost of each node in the tree. In a conventional approach, the code that implements these functions is a single block that must take into account the characteristics that are particular to the computation of the cost of each node (e.g., whether or not the cost of executing leaf nodes is negligible). These differences among the

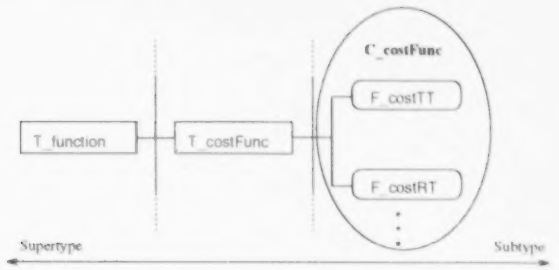


Figure 7.1: Cost model functions as instances of `T_costFunc`

nodes are hard-wired into the cost model function, making it difficult to maintain these functions when new algebraic operators are extended in TIGUKAT.

In order to avoid this problem, a very different approach using the object-oriented paradigm is presented in this research: each algebraic operator node provides individualized behavior for its cost function. This is explained in Section 7.2.1.

In this research, cost model functions are defined as instances of `T_costFunc` (see Figure 7.1). Since `T_costFunc` is a subtype of `T_function`, it inherits all its behaviors (which are described in Section 4.2). For a cost model function, *B_execute* computes the cost model equation whose algorithm is stored in *B_source*. For example, for the equation 7.1 that computes total time of execution, its corresponding implementation is illustrated in Algorithm 7.1. *B_execute* is specialized to return a cost (i.e. an instance of `T_integer`).

For each different cost model equation, a different instance of `T_costFunc` must be created. For example, for the total time cost model function described in Section 7.1.1, the instance *F_costTT* is created as an object of type `T_costFunc` in Figure 7.1. This gives to the cost function component of the optimizer the extensibility property required to incorporate into TIGUKAT query optimizer new cost model functions as they are found useful to measure the actions of a search strategy over the search space. For example, if the response time cost model described in Section 7.1.2 is desired to be incorporated into TIGUKAT query optimizer, the instance *F_costRT* that implements this cost model is created

in the class **C_costFunc** (see Figure 7.1).

**Algorithm 7.1** *Total Time Cost Model Function.:*

*F_costTT* (T_costFuncAlgOp cf, T_algOp *oapt*) : T_integer

**Input:** The cost function cf that is implemented by this function
An OAPT *oapt*

**Output:** The cost of executing cf on *oapt*

```
T_algOp oaptTemp;
T_integer algCost, leftCost, rightCost, totalCost;

{
  algCost = 0;          leftCost = 0;                                  (1)
  rightCost = 0;        totalCost = 0;                                 (2)
  if (oapt != null)                                                    (3)
  {                                                                    (4)
    algCost= oapt.B_cost();                                            (5)
    if ((oapt.B_getOuterRcvr()) != null)                               (6)
    {                                                                  (7)
      leftCost= cf.B_execute(oapt.B_getOuterRcvr());                   (8)
      if ((oapt.B_getInnerArgs()) != null)                             (9)
      {                                                                (10)
        oaptTemp = oapt.B_getInnerArgs().B_getFirst();                 (11)
        while not (oaptTemp == outOfBound)                             (12)
        {                                                              (13)
          rightCost = rightCost + (cf.B_execute(oaptTemp));            (14)
        oaptTemp = (oapt.B_getInnerArgs()).B_getNext();                (15)
        }                                                              (16)
      }                                                                (17)
    }                                                                  (18)
  }                                                                    (19)
  totalCost= algCost + leftCost + rightCost;                           (20)
  return(totalCost);                                                   (21)
}
```

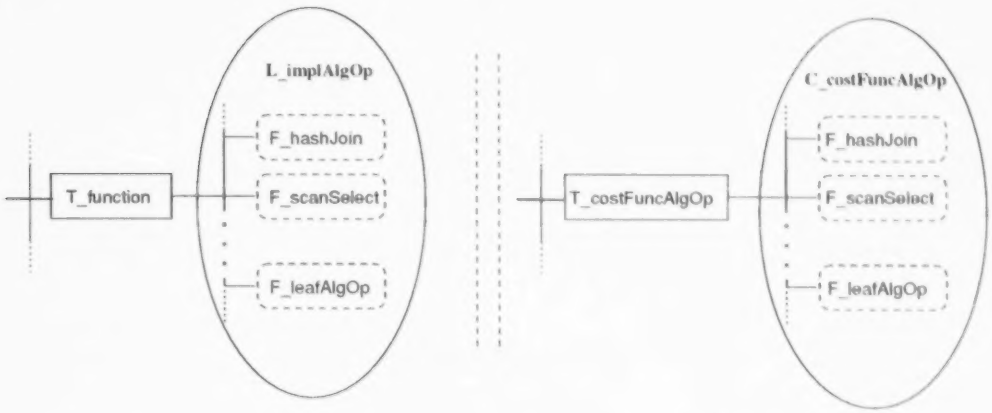


Figure 7.2: Collection **L_implAlgOp** and class **C_costFuncAlgOp**.

### 7.2.1 Algebraic Node Cost Functions

Each algebraic operator context node (of type `T_algOp`) has a different cost function because the algebraic node cost function incorporates optimization issues that potentially vary among the operators. For example, the union operator only needs the cost of accessing the instances of the collections $C_i$, $C_j$ involved in the operation, while the select operator requires the cost of accessing the instances of the collection $C$, in the presence of a predicate $f$. This means that for each algebraic operation implementation (an instance of `T_function`), there is a corresponding cost function (an instance of `T_costFuncAlgOp`) that computes the cost of executing that operation. Then, for each instance in the collection **L_implAlgOp** (that was defined in Section 5.1), there is an instance in the class **C_costFuncAlgOp** (associated to the type `T_costFuncAlgOp`) that computes its cost of execution (see Figure 7.2). Instances of `T_costFuncAlgOp` are the so-called *anc* objects.

Because OAPT nodes are objects of type `T_algOp` which is itself a subtype of `T_function`, they inherit the behavior *B_costFunction* from `T_function`. The application of this behavior to an OAPT node $o$ returns the function object *anc* of type `T_costFuncAlgOp` that implements the computation of the cost of executing the algebraic operation that the node $o$ is representing.

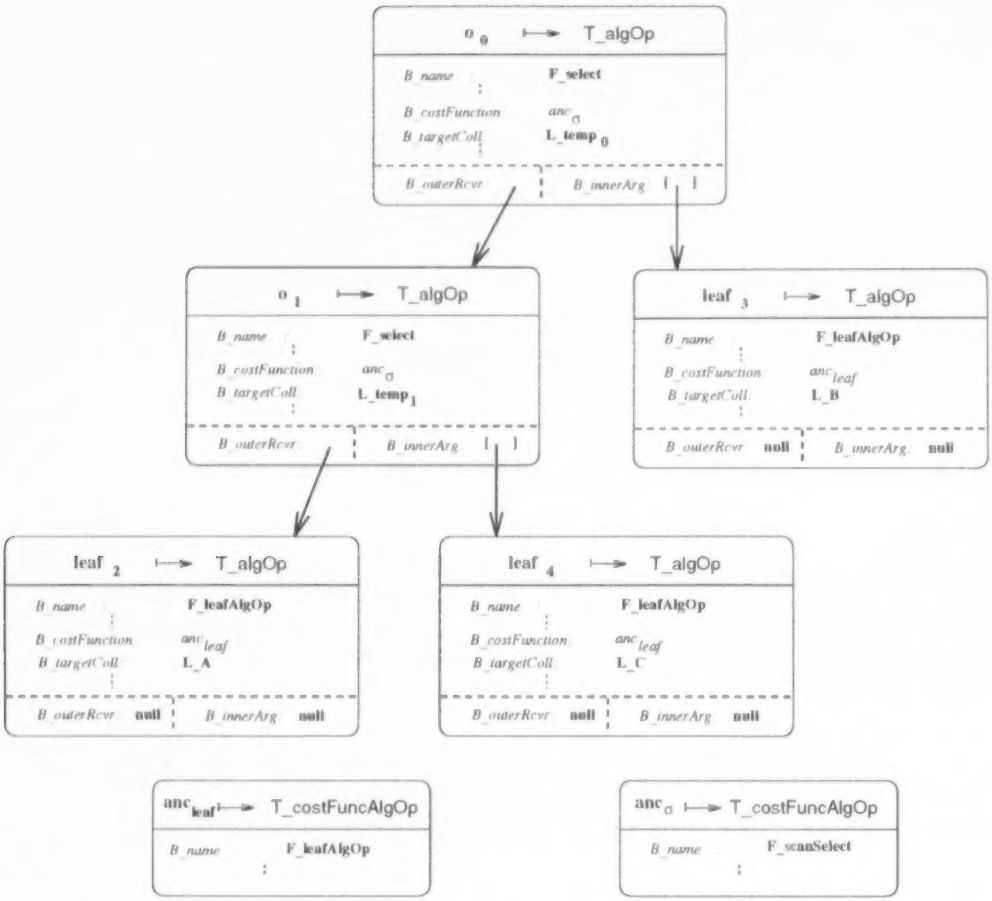


Figure 7.3: OAPT annotated with algebraic node cost functions.

The fundamental advantage of this approach is that each algebraic operator node provides individualized behavior for its cost function. For example, if the cost for some algebraic operators is considered negligible, it is easy to make the cost functions associated to those algebraic nodes to return a constant value (i.e. zero) without having to modify the implementation of the cost model function to consider these exceptions. This gives more flexibility to the optimizer to be able to extend the cost function component for new algebraic operators as they are incorporated in the object algebra.

In order to provide individualized behavior to each of the nodes in the OAPT

| Native Behaviors | Semantics |
|---|---|
| *B_cardinality*(*o*) | Returns an estimated cardinality of the collection object *o*. |
| *B_calcCard*(*o*) | Calculates the cardinality of the collection object *o*. As a side effect, it updates *B_card*. |
| *B_instSize*(*o*) | Returns an estimated size in bytes of an instance in the collection object *o*. |
| *B_calcInstSize*(*o*) | Calculates the size in bytes of an instance in the collection object *o*. As a side effect, it updates *B_instSize*. If the instances are collections, failure is returned(i.e., returning -1). Then, the behavior applies *B_calcInstSize* to each object in the collection *o* (recursively) asking for its instance size and cardinality in order to compute the average size in bytes of an instance. |

Table 7.1: Behavioral summary of statistics defined on `T_collection` type.

(instances of `T_algOp`), each node is annotated with a cost function by filling the behavior *B_costFunction* with the proper cost function object that computes the cost of executing its corresponding algebraic operator. This is done when the OAPT node is first created.

Figure 7.3 shows an OAPT annotated with cost function $anc_{op}$. The first information in the box represents an object instance reference and the mapping to its type. Then, the behaviors that are relevant to the discussion on the cost component for TIGUKAT query optimizer are listed. In the different nodes, *B_costFunction* references the $anc_{op}$s that are cost function instances of type `T_costFuncAlgOp` (that are shown in the bottom of the figure, below the OAPT). The $\mathbf{L\_temp}_i$ is a reference to a temporary collection object and **L_A**, **L_B**, and **L_C** are references to base collection objects.

The algebraic node cost function *anc* is described by equations that are defined by combining costs derived by the OM interface for calls to its interface, and statistics on collections. Because the cost interface that the object manager must provide has not been defined yet, it is not possible in this research to give examples of equations that describe some of the possible algebraic node cost functions. Statistics on collections are defined as behaviors in the interface of `T_collection` as listed in Table 7.1.

An OM interface is defined in [43] that provides a lower level of abstraction

than that provided by the object model and object algebra. The access plan generation is treated as the mapping of object algebra expressions (OAPTs in TIGUKAT) into the new abstraction interface. One of the primary concerns in [43] is to decompose the object algebra operators (i.e. select, map, and so on) into a sequence of simpler operations provided by the interface of an object manager system. The other concern is to respect the encapsulation provided by behaviorally defined objects. This means that the object manager is the only entity that knows how objects are stored. Therefore, [43] assumes that the object manager is capable of derive costs for calls to its interface. However, the operations that can be used to obtain these derived costs are not defined.

In this thesis, we make a similar assumption to that one in [43] in the sense that the object manager is capable of derive costs for calls to its interface. For example, if the OM interface provides a function call *accessInstPred(o,p)* that returns the instances of collection $o$ that satisfy the predicate $p$, it could be possible to ask to the OM interface to derive the cost for this function call (i.e. by calling a function such as *access_cost*(o,p)) as defined in [25]). In order to derive this cost, the object manager can use information such as the existence of an index on a behavior of $o$ that is referenced by the predicate $p$. The existence of this index helps in selecting the appropriate access method to access instances of the collection $o$ (i.e. using the index or performing a sequential scan).

Currently, there is no OM interface defined for TIGUKAT OODBMS. The definition of this interface for TIGUKAT including derived costs for calls to its interface is not within the scope of this research.

### 7.2.2 Execution of Cost Model Functions

In order to compute the cost model function *cf* for a node $o$, the node $o$ collaborates with the function object *cf* by computing its own cost. This interaction is described below step by step and the relevant statements in the Algorithm 7.1 are referenced by number of line in order to clarify the discussion,

1. Execution of the Cost Model Function at the root node of the OAPT. The search strategy component of the optimizer executes the cost model function *cf* passing the root of the tree that is being optimized as a parameter. The root of the tree is denoted as node $o_0$. This execution is performed by the behavioral application

   $cf.B\_execute(o_0)$

   % third line of implementation for *B_initSS* in Table 6.3

   and *cf* is calculated by the behavioral application

   $cf = q.B\_getCostModelFunc()$

   Next, the execution of *cf* fires the computation of the cost for the algebraic operation on node $o_0$ (see step (2)), and the computation of the cost model function on the children of node $o_0$ (see step (3)).

2. Computing the Algebraic Node Cost on node $o_0$. The node $o_0$ knows how to calculate itself the algebraic node cost for the algebraic operation that the node implements. This cost is calculated by applying *B_cost* to the node $o_0$ as illustrated in line **(5)** of Algorithm 7.1. *B_cost* executes the function object $anc_0$ that is returned by applying *B_getCostFunction* to the node $o_0$. This step is performed by the behavioral application

   $anc_0.B\_execute(o_0)$

   and $anc_0$ is calculated by the behavioral application

   $anc_0 = o_0.B\_getCostFunction()$

   The computation of the algebraic node cost on node $o_0$ includes different optimization issues such as statistics on collections.

3. Computing the Cost Model Function for the children of node $o_0$.

   The cost model function *cf* recursively computes the cost of node $o_0$'s children by executing itself passing $o_0$'s children as parameters as illustrated in the path expressions given below,

$cf.B\_execute(o_0.B\_outerRcvr())$ % line (8) of Algorithm 7.1

$foreach_{i=1}^{k}$ $(cf.B\_execute(o_0.B\_innerArgs_i()))$ % line (14) of Algorithm 7.1

4. Returning the cost of executing a node. The costs obtained in steps (2) and (3) are combined according to the cost model equation implemented by the function *cf*, and the result is returned as the cost of executing the node $o_0$. For example, for the total time cost model function defined in Section 7.1, the costs calculated in steps (2) and (3) are summed up and returned by *cf* as the cost of execution of the node $o_0$ (as illustrated in line (20) of Algorithm 7.1).

This process is generalized to be applied to any node $o$ in the OAPT, starting from the root of the tree, node $o_0$, to the leaf nodes where the recursion bottoms out (see lines (9) and (12) in Algorithm 7.1). Then, in a recursive manner, the cost of executing an OAPT is calculated according to its cost model function. This approach follows an object-oriented paradigm to compute the algebraic cost of the individual nodes in the OAPT because the nodes know how to calculate their cost themselves.

Figure 7.4 illustrates this process. The cost model function *cf* that is applied to the root of this OAPT calculates the total time of execution. The values that are returned by the cost model function *cf* after recursively executing itself over each node of the OAPT (that was illustrated in Figure 7.3) are shown on the arrows.

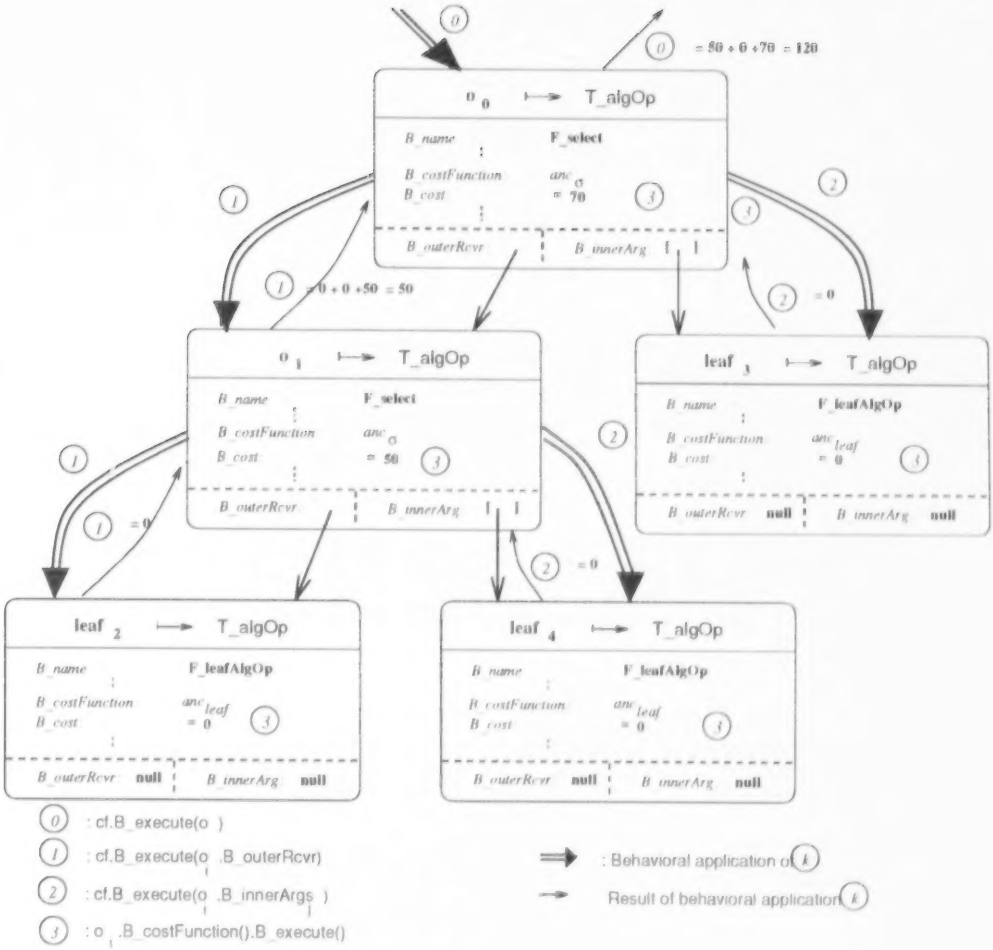
o 0 ⟼ T_algOp
B_name : F_select
B_costFunction : anc σ
B_cost : = 70
B_outerRcvr | B_innerArg
= 50 + 0 +70 = 120
= 0 + 0 +50 = 50
o 1 ⟼ T_algOp
B_name : F_select
B_costFunction : anc σ
B_cost : = 50
B_outerRcvr | B_innerArg
leaf 3 ⟼ T_algOp
B_name : F_leafAlgOp
B_costFunction : anc leaf
B_cost : = 0
B_outerRcvr null | B_innerArg null
= 0
leaf 2 ⟼ T_algOp
B_name : F_leafAlgOp
B_costFunction : anc leaf
B_cost : = 0
B_outerRcvr null | B_innerArg null
leaf 4 ⟼ T_algOp
B_name : F_leafAlgOp
B_costFunction : anc leaf
B_cost : = 0
B_outerRcvr null | B_innerArg null
0 : cf.B_execute(o 0 )
1 : cf.B_execute(o 1 .B_outerRcvr)
2 : cf.B_execute(o 1 .B_innerArgs )
3 : o 1 .B_costFunction().B_execute()
: Behavioral application of k
Result of behavioral application k

Figure 7.4: Execution of the total time cost model function *cf* on an OAPT.

# Chapter 8

# Implementation of TIGUKAT Query Optimizer

This query optimizer is defined as an extension to TIGUKAT object model by using the object-oriented concepts of subtyping and specialization. We call TIGUKAT core object model to the minimum semantically complete object model. It is intended to be powerful enough to support complex extensions such as those that the query optimizer requires. There is a current implementation for the TIGUKAT core object model that consists of libraries of function calls. We refer the reader to [18] for further details on this implementation.

Different alternatives were considered to implement the TIGUKAT query optimizer. A natural way to implement this optimizer is to first extend the core object model and then use an object-oriented programming language (OOPL) built on top of TIGUKAT core object model implementation. However, an OOPL has not yet been defined for TIGUKAT. Another alternative to implement the optimizer is to use an existent OOPL (i.e. C++) to call the functions provided by the core object model implementation. However, because this current implementation is a pre-beta version, it is not robust enough to be used as a platform to implement the query optimizer. Therefore, a third approach that implements the query optimizer as well as the primitive type system by using an existent OOPL was chosen

in this research. This alternative requires the definition of a mapping between the concepts provided by the conceptual object model and the type system provided by the OOPL. This mapping is defined in Sections 8.1 and 8.2.

The query optimizer must be able to access information about the objectbase schema during the optimization process. The conceptual model specifies that the objectbase schema is self-contained in the type system. However, because an interface between an existent OOPL (i.e. C++) and the core object model implementation does not currently exist, the current query optimizer implementation defines internal structures to store information about the objectbase schema.

C++ was selected as the OOPL to implement TIGUKAT query optimizer. There were several reasons for this choice as listed below;

1. C++ is an OOPL that supports the concepts of abstract data types, encapsulation, class hierarchies with inheritance and polymorphism. These features are essential in building an extensible object oriented query optimizer. The components of the TIGUKAT conceptual object model and of the query optimizer can be mapped to C++ under certain restrictions that are mentioned later on in this chapter.

2. C++ provides good programming concepts such as information hiding, modularity, code reusability, and extensibility. These are also important features for the integration of the different modules in which TIGUKAT OBMS has been divided such as the TQL parser, the query optimizer, and the object manager. This constitutes an important reason for the choice of C++ as the language to implement TIGUKAT subsystems [18, 27].

3. C++ implements object-oriented concepts without compromising the efficiency of C language. That makes C++ one of the most efficient object-oriented programming languages [14]. This is a very important consideration when selecting the language to implement the query optimizer where efficiency is essential. Besides, C++ retains the portability of C which is a desired feature for future portability to different platforms.

However, there are some strong differences between C++ and TIGUKAT type system that constrain the implementation of the object model to adhere strictly to the conceptual model. These differences are listed below;

1. TIGUKAT object model supports dynamic schema evolution that allows changes to existing type definitions, creation of new types and classes, and changes to the class hierarchy at run-time. In contrast, C++ is a staticly typed language. This means that when class definitions are changed, they must be recompiled. Therefore, using C++ for implementing the query optimizer leads to a static objectbase definition for the optimizer which is not a problem because the optimizer does not require creation of types or classes on the fly. Besides, the definition of the components of the optimizer (i.e. `T_algOp`) must not be changed while a query is being optimized.

2. In C++, a class is a template to specify instances of this class, but it is not available at run-time. In contrast, TIGUKAT clearly separates the concepts of type and class. It defines a class as a container that keep the instances of a type, and type is a template that contains the specification of objects (behaviors and their associated implementation). This type information is available at run-time. Then, it is clear that C++ programming language and TIGUKAT OBMS differ on the dynamics of class and type definition, respectively.

This thesis is intended to show the viability of an extensible query optimizer as described in Chapter 4. The experimentation consisted in the optimization of some query examples by giving to the query optimizer the corresponding query object annotated with the equivalent initial OAPT as input, and returning an optimized OAPT as output.

In order to support these experiments, the subset of the TIGUKAT type system that is relevant to the query optimizer as well as the query optimizer architecture were implemented using C++ (GNU's C++ implementation called g++) on a Sun SPARC station IPX under UNIX. The query optimizer implementation

contains about 5000 lines of C++ code.

The implementation of the query optimizer architecture includes the implementation of the OAPTs that constitutes the search space, a subset of algebraic transformation rules, an algebraic heuristic search strategy and a total time cost model function.

An algebraic heuristic search strategy (defined in Section 6.1.1) was chosen to be implemented to control the actions over the search space (OAPTs). However, the framework for defining search strategies is general enough to allow the implementation of enumerative or randomized search strategies by extending the `T_searchStrat` hierarchy.

A total time cost model function (defined in Section 7.1.1) was implemented to show the feasibility of modeling the cost function components as objects. For this implementation, the algebraic cost functions were chosen to return a constant value because of the lack of an object manager that derives costs for call to its interface corresponding to those algebraic operators. The search strategy component was extended to support a cost-controlled heuristic search strategy (defined in Section 6.1.2) to illustrate the use of the cost function component by the query optimizer.

## 8.1 Mapping of TIGUKAT Object Model to C++

In C++, a *class* is a user-defined type. The only way to have access to objects of a class is by a set of functions declared as part of the class. These functions are called member functions [44].

A standard methodology to store the specification of classes in C++ is to keep the declaration of a class including the declaration of its member functions in a file .h and to keep the definition of the member functions in a file .C. These files are named with the class' name. For example, the declaration of the class A is stored in a file called A.h, while the definition of its member functions is stored in a file called A.C. We follow this standard when implementing the classes for

the query optimizer.

The mapping of each primitive object in TIGUKAT core object model to C++ is described below.

### 8.1.1 Objects

TIGUKAT supports *strong object identity* (see Section 3.1) meaning that every object has a unique existence within the system that is a feature provided by C++ as well. Conceptually, every TIGUKAT object is a *composite object* meaning that every object has references to other objects. This concept is implemented in C++ by modeling TIGUKAT references to objects as C++ pointers to objects.

### 8.1.2 Types

Each non-atomic type relevant to the query optimizer in the TIGUKAT extended object model is mapped to a corresponding C++ class that is declared in a .h file[1]. These classes are named by `T_` < *typename* >. For example, for the TIGUKAT type `T_object`, the C++ `T_object` class is created. On the other hand, the TIGUKAT atomic types are directly mapped to the corresponding C++ types. For example, `T_integer` is mapped to *int*.

### 8.1.3 Type Hierarchy

The root of the C++ class hierarchy is the `T_object` class which corresponds to the `T_object` type in TIGUKAT Type System. The C++ class hierarchy follows the TIGUKAT type hierarchy by using the C++ mechanism of subclassing, in an analogous way to the TIGUKAT mechanism of subtyping. When member functions are overloaded in a subclass, the C++ mechanism of declaring virtual member functions is used in the respective superclass. Currently, the type hierarchy extended by the query optimizer only requires single inheritance.

[1]In general, OBMSs that support C++ use the .h files as the database schema definition.

### 8.1.4 Collections and Classes

TIGUKAT `T_collection` and `T_class` are implemented by the C++ classes `T_collection` and `T_class`, respectively.

In TIGUKAT, every type that supports instantiation is associated to a class object that manages the instances of that particular type. This is implemented by defining the member function *B_classOf* in the interface of the C++ `T_type` class. This member function is defined as a pointer to an instance of type `T_class`.

In TIGUKAT, object creation is done by applying the behavior *B_new* to the class corresponding to the type of the object. In contrast, in C++, object creation is done by using special member functions called *constructors*. Then, in the current implementation, when a constructor is used, as part of its definition, code is included to make sure that the object is stored in the corresponding class to adhere to TIGUKAT conceptual model in that objects cannot exist without an associated class and classes are automatically maintained by the system.

### 8.1.5 Behaviors and Functions

In TIGUKAT, *behaviors* specify the semantics of an operation, while *functions* implement the semantics of behaviors. A behavior defined on the interface of a type must be explicitly associated with an implementation by the application of the behavior *B_associate*.

In the C++ implementation, a behavior defined in the interface of a TIGUKAT type is mapped to a member function declaration on the corresponding C++ class (in file .h), while the implementation of the behavior is modeled as the definition of the C++ member function (in file .C). Then, in C++, the association of a behavior with its implementation is implicitly done when the respective member function is defined. The code in the body of the definition of the member functions is written in C++.

### 8.1.6 Behavioral and Implementation Inheritance

The query optimizer uses the C++ implementation inheritance mechanism that makes use of virtual tables to handle overloading of function members. In contrast, the TIGUKAT core object model implementation uses the cache table mechanism to solve overloading and late binding of implementations to behaviors at run-time.

## 8.2 Mapping of TIGUKAT Query Optimizer to C++

In order to extend the TIGUKAT base object model for query optimization purposes, the C++ mechanisms of deriving classes (for modeling TIGUKAT subtyping) and defining virtual member functions (for overloading of TIGUKAT behaviors) are used.

The mapping of the components of the TIGUKAT query optimizer to C++ is described below.

### 8.2.1 Search Space

The class `T_algOp` is defined as a derived class of `T_context`, which in turn is defined as a derived class of `T_function`. In order to implement the type `T_algEqRule`, the C++ class `T_algEqRule` is defined as a derived class of `T_rule` which in turn is a derived class of `T_object`.

### 8.2.2 Search Strategy

The class `T_searchStrat` is defined as a derived class of `T_function`. The code for the search algorithm and for the behaviors defined on this type are written in C++ in a way that allows a straightforward translation to a behavioral TIGUKAT OOPL when it is defined and implemented. Currently, there is no programming language for the TIGUKAT system.

### 8.2.3 Cost Model Functions

The C++ capability of declaring member functions of type *function call*, allows to manipulate the address of the function to be executed. However, the declaration of this type of member functions requires a fixed number and type of parameters to be specified because of the static typed nature of C++. This feature is used in the implementation of the cost functions for algebraic operator context nodes.

A cost function object of class `T_costFAlgOp` is stored in the member function *B_costFunction* that is defined in the interface of the algebraic operator context nodes of class `T_algOp`. This cost function must be executed in order to obtain the cost for that node. Since the executable code that implements these cost functions may potentially differ for each algebraic operator, but they have the same number and type of parameters (`T_algOp`), and the same return type (`T_integer`), the C++ capability of declaring a member function of type function call is used to declare the behavior *B_executable* for them.

# Chapter 9

# Conclusions

This thesis describes an extensible query optimizer architecture for OBMSs. The identifying characteristic of this design is the use of the object-oriented philosophy in providing extensibility. The architecture defines all components of the optimizer (search space and transformation rules, cost function, and cost strategies) as well as the queries themselves as first-class objects. This is consistent both with the TIGUKAT object model and the object-oriented design philosophy. In a sense, this is using the medicine that is normally prescribed to others. The end result, which we believe to be a significant advantage, is that both the query model and the query optimizer become direct extensions of the TIGUKAT object model which can be managed (stored, changed, queried) just like any other object.

In order to implement the query optimizer, a subset of the TIGUKAT core object model relevant to it was implemented using C++. This requires a mapping between TIGUKAT and C++ type systems. The core object model was extended by using the C++ mechanisms of deriving classes (for modeling subtyping) and defining virtual member functions (for overloading of behaviors) in order to support the extended object model that is required for query optimization purposes, implementing in this way the TIGUKAT query optimizer. However, because of the differences between TIGUKAT and C++ in the dynamics of class and type definition, as well as in their dynamic and static type nature, respectively, the

current implementation loses the full uniformity feature provided by TIGUKAT object model.

This implementation constitutes a prototype for the query optimizer that allowed us to prove that the TIGUKAT query optimizer can be built as an extension to the TIGUKAT core object model.

## 9.1 Future Research

Although the modeling of the cost component of the optimizer is described in this thesis (Section 7.2.1), further research must be done in the cost information that the object manager can provide to the optimizer when using cost-controlled search strategies. The definition of an interface to the object manager for TIGUKAT object model is required. The work done in [43] can be used as a basis for defining TIGUKAT object manager interface.

Further research must also be done in defining semantic transformation rules for TIGUKAT query optimizer. For example, a possible semantic transformation rule could be that if the class $C_2$ is a subclass of $C_1$, the intersection of those two classes, $C_1$ and $C_2$, produces the class $C_2$ as a result because the object model restricts each object to belong to only one class.

Future research must be done in defining a methodology to select the "best" execution algorithms that implement each algebraic node in the OAPT based on physical information provided by the object manager such as indexes, clustering, and so on. The selection of these algorithms is done by the execution plan generator[1] providing the object manager can derive physical information through function calls that belong to its interface.

Other topics of future research are the exploration of new techniques for optimization of behaviors inside queries and the type inferencing mechanism for transformation rules that involve target-creating algebraic operators.

[1]The execution plan generator is modeled by the behavior *B_genExecPlan* that is defined in the interface of the type `T_query`.

# Bibliography


[1] S. Aronoff. *Geographic Information Systems: A Management Perspective*. WDL Publications, 1989.

[2] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the $O_2$ object-oriented database system. In *Proc. 2nd. Int. Workshop on Database Programming Languages*, pages 122–138, 1989.

[3] J.A. Blakeley, W.J. McKenna, and G. Graefe. Experiences building the Open OODB query optimizer. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 287–296, 1993.

[4] S. Cluet. *Langages et Optimisation de Requetes pour Systemes de Gestion de Base de donee oriente-objet*. PhD thesis, Universite de Paris-Sud, 1991.

[5] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 383–392, 1992.

[6] E. F. Codd. Relational completeness of data base sublanguages. In *Courant Computer Science Symposium on Data Base Systems*, volume 6, pages 65–98. Prentice-Hall, May 1971.

[7] U. Dayal, A. Buchmann, and D. McCarthy. Rules are objects too: A knowledge model for an active object-oriented database system. In *Proc. 2nd Int. Workshop on Object-Oriented Database Systems*, pages 129–143, 1988.

[8] A. Dominguez. Query optimization in multidatabase systems. Master's thesis, University of Alberta, Edmonton, Alberta, Canada, 1993.

[9] J.C. Freytag. A rule–based view of query optimization. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 173–180, 1987.

[10] A.V. Gelder and R.W. Topor. Safety and Translation of Relational Calculus Queries. *ACM Transactions on Database Systems*, 16(2):235–278, June 1991.

[11] G. Graefe and D.J. DeWitt. The EXODUS optimizer generator. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 160–172, 1987.

[12] G. Graefe and D. Maier. Query optimization in object-oriented database systems: A prospectus. In *Proc. 2nd Int. Workshop on Object-Oriented Database Systems*, pages 358–363. Springer Verlag, 1988.

[13] G. Graefe and W.J. McKenna. The Volcano optimizer generator. In *Proc. 9th Int. Conf. on Data Engineering*, pages 209–218, 1993.

[14] R. Gupta. A quickstart introduction to C++. In R. Gupta and E. Horowitz, editors, *Object-Oriented Databases with applications to CASE, Networks and VLSI CAD*, pages 324–342. Prentice Hall, 1991.

[15] W. Hasan and H. Pirahesh. Query rewrite optimization in Starburst. Technical Report TR RJ 6367, IBM Alamden Research Center, August 1988.

[16] Y. Ioannidis and Y. Cha Kang. Randomized algorithms for optimizing large join queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 312–321, 1990.

[17] Y. Ioannidis and E. Wong. Query optimization by simulated annealing. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 9–22, 1987.

[18] B.B. Irani. Implementation of the TIGUKAT object model. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report, TR93–10.

[19] B.P. Jenq, D. Woelk, W. Kim, and W-L. Lee. Query processing in distributed ORION. In *Advances in Database Technology — EDBT'90*, pages 169–187. Springer Verlag, 1988.

[20] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 148–157, 1991.

[21] S.N. Khoshafian and G.P. Copeland. Object Identity. In *OOPSLA '86 Conference Proceedings*, pages 406–416, 1986.

[22] W. Kim. A model of queries for object-oriented databases. In *Proc. 15th Int. Conf. on Very Large Data Bases*, pages 423–432, 1989.

[23] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. 12th Int. Conf. on Very Large Data Bases*, pages 128–137, 1986.

[24] R. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *Proc. 17th Int. Conf. on Very Large Data Bases*, pages 363–373, 1991.

[25] R. Lanzelotte, P. Valduriez, and M. Zait. Optimization of object-oriented recursive queries using cost-controlled strategies. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 256–265, 1992.

[26] R. Lanzelotte, P. Valduriez, M. Ziane, and J.-P. Cheiney. Optimization of nonrecursive queries in OODBs. In *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, volume 566 of *Lecture Notes in Computer Science*, pages 1–21. Springer Verlag, 1991.

[27] A.P. Lipka. The design and implementation of TIGUKAT user languages. Master's thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993. Available as University of Alberta Technical Report, TR93–11.

[28] G. Mitchell, U. Dayal, and S.B. Zdonik. Control of an extensible query optimizer: A planning-based approach. In *Proc. 19th Int. Conf. on Very Large Data Bases*, pages 517–528, 1993.

[29] G. Mitchell, S.B. Zdonik, and U. Dayal. An architecture for query processing in persistent object stores. In *Proceedings of the Hawaii International Conference on System Sciences*, volume II, pages 787–798, 1992.

[30] G. Mitchell, S.B. Zdonik, and U. Dayal. Optimization of object-oriented queries: Problems and approaches. In A. Dogac, M. T. Özsu, A.Biliris, and T. Sellis, editors, *Advances in Object-Oriented Database Systems*. Springer Verlag, 1994. (forthcoming).

[31] J. Orenstein, S. Haradvala, B. Margulies, and D. Sakahara. Query processing in the ObjectStore database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 403–412, 1992.

[32] M. T. Özsu, R. J. Peters, B. Irani, A. Lipka A. Muñoz, and D. Szafron. TIGUKAT Object Management System: Initial design and current directions. In *Proc. of CASCON'93 Conf.*, pages 595–611, Oct 1993.

[33] M.T. Özsu and J. Blakeley. Query processing in object-oriented database systems. In W. Kim, editor, *Database Challenges in the 1990s*. Addison-Wesley/ACM Press, 1994. (forthcoming).

[34] M.T. Özsu, U. Dayal, and P. Valduriez. An introduction to distributed object management. In M.T. Özsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, pages 1–24. Morgan Kaufmann, 1994.

[35] R. J. Peters. *TIGUKAT: A Uniform Behavioral Objectbase Management System*. PhD thesis, Department of Computing Science. University of Alberta, 1994. (forthcoming).

[36] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. An extensible query model and its languages for a uniform behavioral object management system. In *Proc. Second Int. Conf. on Information and Knowledge Management*, pages 403–412, November 1993. A full version of this paper is available as University of Alberta technical report TR93-01.

[37] R.J. Peters, A. Lipka, M.T. Özsu, and D. Szafron. The query model and query language of TIGUKAT. Technical Report TR93-01, Department of Computing Science, University of Alberta, January 1993.

[38] R.J. Peters and M.T. Özsu. Reflection in a Uniform Behavioral Object Model. In *Proc. 12th Int. Conf. on Entity-Relationship Approach*, pages 37–49, December 1993.

[39] R.J. Peters, M.T. Özsu, and D. Szafron. TIGUKAT: An object model for query and view support in object database systems. Technical Report TR92-14, Department of Computing Science, University of Alberta, October 1992.

[40] R.E.Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[41] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, and D. Beech. Third-generation data base system manifesto. *ACM SIGMOD Record*, 19(3):31–44, September 1990.

[42] D.D. Straube and M.T. Özsu. Queries and query processing in object-oriented database systems. *ACM Transactions on Information Systems*, 8(4):387–430, October 1990.

[43] D.D. Straube and M.T. Özsu. Query optimization and execution plan generation in object-oriented data management systems. *IEEE Transactions on Knowledge and Data Eng.*, (in press), 1992. (A short version appears in *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases, 1991*).

[44] B. Stroustrup. *The C++ Programming Language (2nd. edition).* Addison Wesley, 1992.

[45] A. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 367–376, 1989.

[46] A. Swami and A. Gupta. Optimization of large join queries. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 8–17, 1988.

[47] C.D. Tomlin. *Geographic Information Systems and Cartographic Modeling.* Prentice-Hall, 1990.

[48] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, second edition 1983.

# Appendix A

# Walk Through the Optimizer Architecture by Example

The following example illustrates a possible query on the GIS. They are first expressed in TQL, then the corresponding object calculus expression is given, and finally, the equivalent algebraic expression is shown. In the algebraic expressions, we subscript an operand collection by the variable which ranges over it, and use '$\leftarrow$' to denote assignment of intermediate and final results.

All TQL queries are either submitted from within a programming language (embedded TQL) or during a user session. Since embedded TQL is not yet available, only queries submitted during a user session are considered in this example. A simple session control language called TIGUKAT Control Language (TCL), that controls the creation of the appropriate objects and interprets the optimization commands is defined in [37].

**Example A.1** Return the maps which show the areas where retired people are living.

TQL statement:

**select** $o$
**from** $o$ **in C_map**
**where exists** ( **select** $p$
**from** $p$ **in C_person**, $q$ **in C_dwelling**
**where** ($p.B\_age() \geq 65$ **and** $q = p.B\_residence()$
**and** $q.B\_inzone() \in o.B\_zones()$))

Calculus formula:

$$\{\ o \mid \mathbf{C\_map}(o)\ \wedge\ \exists p(\mathbf{C\_person}(p)\ \wedge\ \exists q(\mathbf{C\_dwelling}(q) \wedge\ p.B\_age \geq 65\ \wedge\ q = p.B\_residence\ \wedge\ q.B\_inzone \in o.B\_zones))\}$$

Algebraic expression:

$$Result \leftarrow \mathbf{C\_map}_o\ \sigma \left[ \begin{array}{c} p.B\_age \geq 65\ \wedge \\ q = p.B\_residence\ \wedge \\ q.B\_inzone \in o.B\_zones \end{array} \right] < \mathbf{C\_person}_p,\ \mathbf{C\_dwelling}_q >$$

Behavioral algebra expression:

$$\mathsf{resultQ3} \leftarrow \mathbf{C\_map}.B\_select(f, [\mathbf{C\_person}, \mathbf{C\_dwelling}])$$

Translation into object algebra processing tree (a context):

$$\mathsf{oapt_o} \leftarrow (\mathbf{C\_map}.B\_select(f, [\mathbf{C\_person}, \mathbf{C\_dwelling}])).B\_context$$

1. Query object creation. The TCL interpreter creates the query (say **q1**) and then sets the TQL statement of the query as the source of this query.

   **q1** $\leftarrow$ **C_query**.*B_new*
   **q1**.*B_setSource*(TQL_statement)

   Note that in TIGUKAT, for every (conceptual) behavior *B_behavior* whose value can be changed by users, a pair *B_setBehavior*/*B_getBehavior* (set value / get value) behaviors are defined.

2. Search strategy specification. The architecture allows the user (or the application submitting the query) to set the search strategy and the behaviors associated with it. In this case, we assume that the system defaults are used:

   **q1**.*B_setSearchStrat*(*F_heurSS*)

3. Cost model specification. The architecture allows the user (or the application submitting the query) to set the cost model and the behaviors associated with it. In this case, we assume that the system defaults are used:

   **q1**.*B_setCostModelFunc*(*F_costTT*)

   The cost model is only required when the search strategy is cost-controlled.

4. Compilation of query object. The query is compiled by applying the behavior *B_compile* to query **q1**:

   **q1**.*B_compile*()

   The side-effect of applying the *B_compile* behavior is the following:

   (a) Parsing and calculus-algebra translation

   **oapt**$_o$.*B_setName*(F_selAlgOp) (1)
   **oapt**$_o$.*B_setRcvrType*(T_class) (2)
   **oapt**$_o$.*B_setArgTypes*([T_class,T_class]) (3)
   **oapt**$_o$.*B_setResultType*(T_collection) (4)
   **oapt**$_o$.*B_setRcvMbrType*(T_map) (5)
   **oapt**$_o$.*B_setArgMbrTypes*([T_person, T_dwelling]) (6)
   **oapt**$_o$.*B_setResultMbrType*(T_map) (7)
   **oapt**$_o$.*B_setTargetVar*($o$) (8)

$\mathtt{oapt}_o.B\_setConstraint(f)$ (9)

$\mathtt{q1}.B\_setInitialOAPT(\mathtt{oapt}_o)$ (10)

$\mathtt{q1}.B\_setResult(\mathtt{null})$ (11)

The result of the translation of the calculus query into an algebra expression is the generation of an OAPT and setting various behaviors (expressions (1) – (9) above). By and large these expressions are self-explanatory. The ones that require some explanation are (9)–(11). The $f$ in expression (9) is a reference to an object of type `T_formula` which represents the predicate of the selection operator. Statements (10) and (11) set the two behaviors of the query object `q1`. The result of the query is null at this point since it has not yet been executed.

(b) Algebraic optimization

The second major action resulting from the compilation of a query is its optimization. This is accomplished by the application of *B_optimize* to `q1`:

$finalOAPT \leftarrow \mathtt{q1}.B\_optimize()$

The *B_optimize* behavior carries out plan optimization on $\mathtt{q1}.B\_initialOAPT$ using the search strategy $\mathtt{q1}.B\_searchStrat$. This behavior implements the control strategy for the plan optimizer. The result is an optimal OAPT that is saved:

$\mathtt{q1}.B\_setOptimizedOAPT(finalOAPT)$

this OAPT records the optimal execution plan as part of the query. This is useful both for later executions which do not need to be optimized and for being able to implement operators such as "explain" which informs the requestor of the optimal execution plan that the

optimizer has chosen. These operators are now quite common in state-of-the-art DBMSs.

The code for the function object that implements *B_optimize* is illustrated in Algorithm A.1.

**Algorithm A.1** *Optimizer.:*

```
F_optimize(T_query q): T_list⟨T_algOp⟩
{
    return((q1.B_getSearchSS).B_execute(q1));
}
```

(c) Execution plan generation

This is the last step in the TIGUKAT query processing methodology whereby the algebraically optimized OAPT is submitted to the object manager for further optimization and execution. This part is not in the scope of this research. However, it is planned within the frame of the TIGUKAT OBMS project that this step will follow the execution plan generation method proposed in [43] that generates the query execution plan by replacing each individual algebra operator from the optimized OAPT with a "best" subtree of Object Manager (OM) calls. These object manager calls that are part of the set of low level object manipulation primitives that constitutes the interface to the OM can be modeled in TIGUKAT as function objects.

The execution plan generation is supported in this architecture by providing the behavior *B_genExecPlan*. When *B_genExecPlan* is applied to `q1`:

`q1`.*B_genExecPlan*()

it results in the set of execution plans to be stored as part of the query. These execution plans can be accessed later by the application of *B_execPlanFamily* to query `q1`.

5. Execution of the query object. The execution of the query may be invoked by the user explicitly if the query is already optimized. In the case of ad hoc queries submitted during a user session, the query is executed when it is compiled and optimized. Thus, the TCL interpreter applies the *B_execute* behavior to **q1**.

   **q1**.*B_setResult*(**q1**.*B_execute*())

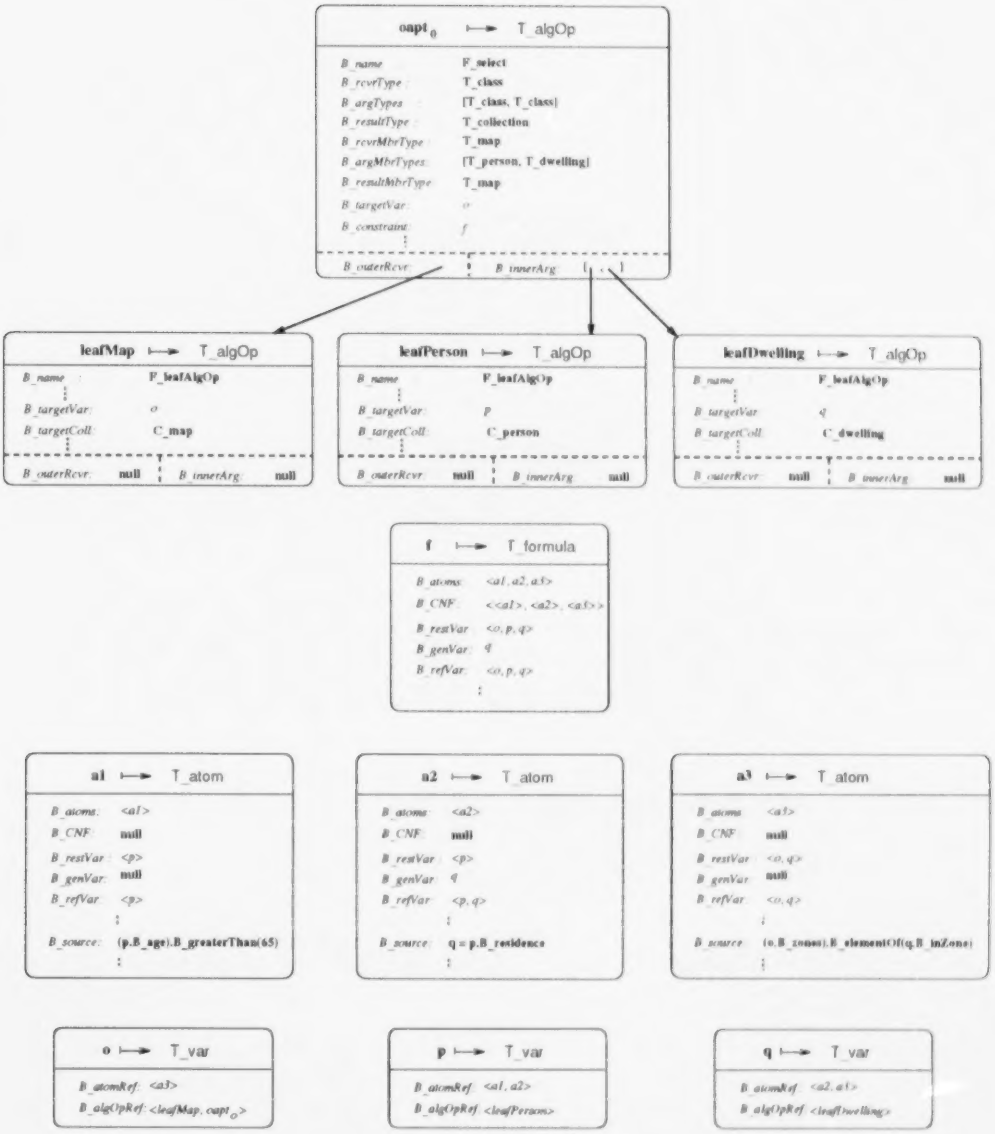


Figure A.1: Object algebra processing tree

# Appendix B

# Extensions to TIGUKAT Type System

In this appendix, we define the full behavioral specification of the extended type system of TIGUKAT to model an extensible query optimizer inside the Object Model. The extended type lattice is shown in Figure 4.1.

In the following specifications, we use variables $o$, $p$ and $q$ in examples as references to objects of various particular types. The example behavioral applications assume left associativity in the absence of qualifying parenthesis. That is, the behavioral application $B\_something(o)(p)$ is equivalent to $(B\_something(o))(p)$.

The type specifications are divided into the following components. The name of the type, its corresponding class, its supertypes, its subtypes, the native behaviors defined by the type and the derived behaviors defined by the type. Native behaviors are those which are introduced by the type (i.e., they are not inherited). Derived behaviors are those which are defined in terms of existing behaviors (i.e., they are not primitive to the type system, but are defined for brevity and ease of use). The implementations for some of the inherited behaviors are refined in the subtypes and their extended semantics are given in the refined behaviors section.

Besides defining new types to model query optimization aspects (i.e., `T_algOp`), we have added new native behaviors to some of the existent types in the type prim-

| Type | Signatures | |
|---|---|---|
| T_object | *B_self*: | T_object |
| | *B_mapsto*: | T_type |
| | *B_conformsTo*: | T_type → T_boolean |
| | *B_equal*: | T_object → T_boolean |
| | *B_notequal*: | T_object → T_boolean |
| T_type | *B_interface*: | T_collection⟨T_behavior⟩ |
| | *B_native*: | T_collection⟨T_behavior⟩ |
| | *B_inherited*: | T_collection⟨T_behavior⟩ |
| | *B_specialize*: | T_type → T_boolean |
| | *B_subtype*: | T_type → T_boolean |
| | *B_subtypes*: | T_collection⟨T_type⟩ |
| | *B_supertypes*: | T_collection⟨T_type⟩ |
| | *B_sub-lattice*: | T_poset⟨T_type⟩ |
| | *B_super-lattice*: | T_poset⟨T_type⟩ |
| | *B_classof*: | T_class |
| | *B_tmeet*: | T_type → T_type |
| | *B_tjoin*: | T_type → T_type |
| | *B_tproduct*: | T_type → T_type |

Table B.1: Behavioral summary of extended non-atomics primitive types for optimization purposes.

itive system (i.e., *B_card* was added to T_collection), and overloaded some of the inherited behaviors (i.e., *B_executable* for T_query). These new and overloaded behaviors related to optimization are specifically shown as separate components in the type specification.

A behavioral summary of the primitive types that have been extended together with the types that have been added to the type lattice for query optimization purposes is given in tables B.1, B.2, B.3 and B.4.

| Type | Signatures | |
|---|---|---|
| T_behavior | *B_name*: | T_string |
| | *B_argTypes*: | T_list⟨T_type⟩ |
| | *B_resultType*: | T_type |
| | *B_semantics*: | T_object |
| | *B_associate*: | T_type → T_function → T_behavior |
| | *B_implementation*: | T_type → T_function |
| | *B_primitiveApply*: | T_object → T_object |
| | *B_apply*: | T_object → T_list → T_object |
| | *B_defines*: | T_collection⟨T_type⟩ |
| | *B_costBehavior*: | T_type → T_list⟨type⟩ → T_real |
| T_function | *B_name*: | T_string |
| | *B_argTypes*: | T_list⟨T_type⟩ |
| | *B_resultType*: | T_type |
| | *B_comments*: | T_string |
| | *B_source*: | T_object |
| | *B_primitiveExecute*: | T_object → T_object |
| | *B_execute*: | T_list → T_object |
| | *B_basicExecute*: | T_list → T_object |
| | *B_compile*: | T_object |
| | *B_executable*: | T_object |
| | *B_costFunction*: | T_list⟨object⟩ → T_costFunc |
| | *B_cost*: | T_list⟨object⟩ → T_integer |
| T_collection | *B_typeof*: | T_type |
| | *B_I*: | T_collection |
| | *B_select*: | T_formula → T_list⟨T_collection⟩ → T_collection |
| | *B_generate*: | T_formula → T_var → T_list⟨T_collection⟩ → T_collection |
| | *B_map*: | T_list⟨T_behavior⟩ → T_list⟨T_collection⟩ → T_collection |
| | *B_project*: | T_collection⟨T_behavior⟩ → T_collection |
| | *B_union*: | T_collection → T_collection |
| | *B_difference*: | T_collection → T_collection |
| | *B_intersection*: | T_collection → T_collection |
| | *B_product*: | T_list⟨T_collection⟩ → T_collection |
| | *B_join*: | T_formula → T_list⟨T_collection⟩ → T_collection |
| | *B_cardinality*: | T_integer |
| | *B_calcCard*: | T_integer |
| | *B_instSize*: | T_integer |
| | *B_calcInstSize*: | T_integer |

Table B.2: Behavioral summary of extended non-atomics primitive types for optimization purposes.

| Type | Signatures | |
|---|---|---|
| T_query | *B_initialOAPT*: | T_algOp |
| | *B_optimizedOAPT*: | T_list⟨T_algOp⟩ |
| | *B_searchStrat*: | T_searchStrat |
| | *B_transformations*: | T_list⟨T_algEqRule⟩ |
| | *B_argMbrTypes*: | T_list⟨T_type⟩ |
| | *B_resultMbrType*: | T_type |
| | *B_optimize*: | T_list⟨T_algOp⟩ |
| | *B_genExecPlan*: | T_collection⟨T_function⟩ |
| | *B_execPlanFamily*: | T_collection⟨T_function⟩ |
| | *B_basicExecSave*: | T_list → T_object |
| | *B_basicExecDontSave*: | T_list → T_object |
| | *B_budgetOpt*: | T_integer |
| | *B_lastOpt*: | T_date |
| | *B_lastExec*: | T_date |
| | *B_result*: | T_collection |
| T_context | *B_outerRcvr*: | T_context |
| | *B_innerArg*: | T_list⟨T_context⟩ |
| | *B_rcvrType*: | T_type |
| | *B_argTypes*: | T_list⟨T_type⟩ |
| | *B_resultType*: | T_type |
| T_algOp | *B_outerRcvr*: | T_algOp |
| | *B_innerArg*: | T_list⟨T_algOp⟩ |
| | *B_rcvrType*: | T_type |
| | *B_argTypes*: | T_list⟨T_type⟩ |
| | *B_resultType*: | T_type |
| | *B_rcvrMbrType*: | T_type |
| | *B_argMbrTypes*: | T_list⟨T_type⟩ |
| | *B_resultMbrType*: | T_type |
| | *B_targetVar*: | T_var |
| | *B_targetColl*: | T_collection |
| | *B_constraint*: | T_object |
| | *B_execAlgorithm*: | T_function |
| | *B_splitLeft*: | T_algOp |
| | *B_splitRight*: | T_algOp |
| | *B_linkLeft*: | T_algOp → T_algOp |
| | *B_linkRight*: | T_algOp → T_algOp |
| | *B_assemble*: | T_algOp → T_list⟨T_algOp⟩ → T_algOp |
| | *B_dissassemble*: | T_list⟨T_object⟩ |
| T_searchStrat | *B_execute*: | T_algOp → T_algOp |
| | *B_initSS*: | T_object |
| | *B_stopCond*: | T_boolean |
| | *B_setNextState*: | T_list⟨T_algOp⟩ → T_algOp |
| | *B_action*: | T_algOp → T_list⟨T_algOp⟩ |
| | *B_goal*: | T_collection⟨T_algOp⟩ |
| | *B_optimal*: | T_algOp |

Table B.3: Behavioral summary of non-atomics types added to the primitive type system for optimization purposes.

| Type | Signatures | |
|---|---|---|
| T_rule | *B_cond*: | T_list⟨T_formula⟩ |
| | *B_checkCond*: | T_object → T_boolean |
| | *B_action*: | T_object → T_object |
| T_algEqRule | *B_leftSideFunc*: | T_algOp → T_function |
| | *B_rightSideFunc*: | T_algOp → T_function |
| | *B_matchLeft*: | T_algOp → T_boolean |
| | *B_matchRight*: | T_algOp → T_boolean |
| | *B_condLeft*: | T_list⟨T_formula⟩ |
| | *B_cond*: | T_list⟨T_formula⟩ |
| | *B_checkCondLeft*: | T_algOp → T_boolean |
| | *B_checkCond*: | T_algOp → T_boolean |
| | *B_actionLeftFunc*: | T_algOp → T_function |
| | *B_actionLeft*: | T_algOp → T_algOp |
| | *B_actionRightFunc*: | T_algOp → T_function |
| | *B_action*: | T_algOp → T_algOp |
| | *B_algExpression*: | T_object |
| T_formula | *B_argTypes*: | T_null |
| | *B_resultType*: | T_type |
| | *B_source*: | T_string |
| | *B_compile*: | T_context |
| | *B_executable*: | T_context |
| | *B_execute*: | T_null → T_boolean |
| | *B_basicExecute*: | T_null → T_boolean |
| | *B_atoms*: | T_list⟨T_atom⟩ |
| | *B_CNF*: | T_list⟨T_list⟨T_atom⟩⟩ |
| | *B_restVar*: | T_list⟨T_var⟩ |
| | *B_genVar*: | T_var |
| | *B_refVar*: | T_list⟨T_var⟩ |
| | *B_splitRestrDisj*: | T_collection |
| | *B_splitRestrConj*: | T_collection |
| T_atom | *B_argTypes*: | T_null |
| | *B_resultType*: | T_type |
| | *B_source*: | T_string |
| | *B_compile*: | T_context |
| | *B_executable*: | T_context |
| | *B_execute*: | T_null → T_boolean |
| | *B_basicExecute*: | T_null → T_boolean |
| | *B_atoms*: | T_list⟨T_atom⟩ |
| | *B_CNF*: | T_null |
| | *B_restVar*: | T_list⟨T_var⟩ |
| | *B_genVar*: | T_var |
| | *B_refVar*: | T_list⟨T_var⟩ |
| T_var | *B_atomRef*: | T_list⟨T_atom⟩ |
| | *B_algOpRef*: | T_algOp |

Table B.4: Behavioral summary of non-atomics types added to the primitive type system for optimization purposes.

# T_object

Supertypes: *none*

Subtypes: T_type, T_collection, T_behavior, T_function, T_atomic

Native Behaviors:

**self**

*B_self* : T_object
Example: *B_self*(*o*)
Symbol: $I_o$

Simply returns the argument object *o*. This is a mathematical *identity* operation for objects.

**mapsto**

*B_mapsto* : T_type
Example: *B_mapsto*(*o*)
Symbol: $o \mapsto$

Returns the singleton type object which was used as a template to create the argument object *o*. Every object in the system has a *mapsto* type.

**conformsTo**

*B_conformsTo* : T_type → T_boolean
Example: *B_conformsTo*(*o*)(*p*)
Symbol: $o \rightsquigarrow p$

If the first argument object *o* conforms to the type argument object *p*, the object **true** is returned. Otherwise **false** is returned.

**equal**

*B_equal* : T_object → T_boolean
Example: *B_equal*(*o*)(*p*)
Symbol: $o = p$

If the first argument object *o* is identity equal to the second argument object *p*, the object **true** is returned. Otherwise **false** is returned.

Derived Behaviors:

**notequal**

*B_notequal* : T_object → T_boolean
Example: *B_notequal*(*o*)(*p*)
Symbol: $o \neq p$
Derivation:
$\neg(o = p)$

If the first argument object *o* is identity equal to the second argument object *p*, the object **false** is returned. Otherwise **true** is returned.

# T_collection

Supertypes: T_object
Subtypes: *none*
Native Behaviors:

**typeof**

*B_typeof* : T_type
Example: *B_typeof*(*o*)
Symbol:

Returns the type object associated with the argument collection object *o*. Every collection is associated with exactly one type object, but a type object may be associated with many collections.

Native Behaviors: Related to Query Optimization.

**I**

*B_I* : T_collection
Example: *B_I*(*o*)
Symbol:

Returns the collection object *o*. This is a mathematical identity operation for the object algebra because of the closure of the algebra on collections.

**select**

*B_select* : T_formula → T_list⟨T_collection⟩ → T_collection
Example: *B_select*(*o*)(*p*)(*q*)
Symbol:

Returns a collection object resulting from applying the algebraic select operator to the collection object *o* with formula *p* and using the collection objects given in the list *q* as arguments.

**generate**

*B_generate* : T_formula → T_string → T_list → T_collection
Example: *B_generate*(*o*)(*p*)(*q*)(*r*)
Symbol:

Returns a collection object resulting from applying the algebraic generate operator to the collection object *o* with formula *p* and using the collection objects given in the list *r* as arguments. The formula *p* must contain one or more generating atoms for target variable *q*.

**map**

*B_map* : T_list⟨T_behavior⟩ → T_list → T_collection
Example: *B_map*(*o*)(*p*)(*q*)
Symbol:

Returns a collection object resulting from applying the sequence of behaviors given in the list *p* to each of the objects that belong to the collection *o* using the collection objects given in the list *q* as arguments.

**project**

*B_project* : T_collection⟨T_behavior⟩ → T_collection
Example: *B_project*(*o*)(*p*)
Symbol:

Returns a collection object containing the collection of objects denoted by the collection object *o* with a new type coinciding with the behavioral specification of *p*.

**union**

*B_union* : T_collection → T_collection
Example: *B_union*(*o*)(*p*)
Symbol:

| Returns a collection object resulting from applying the algebraic union operator to the collection objects $o$ and $q$. |
|---|

**difference**

*B_difference* : T_collection → T_collection
Example: *B_difference*(*o*)(*p*)
Symbol:

| Returns a collection object resulting from applying the algebraic difference operator to the collection objects $o$ (minuend) and $q$ (sutrahend). |
|---|

**intersection**

*B_intersection* : T_collection → T_collection
Example: *B_intersection*(*o*)(*p*)
Symbol:

| Returns a collection object resulting from applying the algebraic intersection operator to the collection objects $o$ and $q$. |
|---|

**product**

*B_product* : T_list⟨T_collection⟩ → T_collection
Example: *B_product*(*o*)(*p*)
Symbol:

| Returns a collection object containing new object lists $(o_1, p_1, ..., p_n)$ resulting from applying the algebraic product operator to the collection object $o$ using the collection objects given in the list $p$ as arguments. |
|---|

**join**

*B_join* : T_formula → T_list⟨T_collection⟩ → T_collection
Example: *B_join*(*o*)(*p*)(*q*)
Symbol:

| Returns a collection object containing new object lists $(o_1, q_1, ..., q_n)$ resulting from applying the algebraic join operator to the collection object $o$ with formula $p$ and using the collection objects given in the list $q$ as arguments. |
|---|

**cardinality**

*B_cardinality* : T_integer
Example: *B_cardinality*(*o*)
Symbol:

| Returns an estimated cardinality of the collection object $o$. It is implemented by a stored function. |
|---|

**calcCard**

*B_calcCard* : T_integer
Example: *B_calcCard*(*o*)
Symbol:

| Calculates the cardinality of the collection object $o$. As a side effect, it updates *B_card*. It is implemented by a computed function. |
|---|

**instSize**

*B_instSize* : T_integer
Example: *B_instSize*(*o*)
Symbol:

| Returns an estimated size in bytes of an instance in the collection object $o$. It is implemented by a stored function. |
|---|

**calcInstSize**

*B_calcInstSize* : `T_integer`
Example: *B_calcInstSize*(*o*)
Symbol:

Calculates an estimated size in bytes of an instance in the collection object *o*. As a side effect, it updates *B_instSize*.
If the instances are collections, the behavior *B_calcInstSize* returns failure (i.e., returning -1). Then, the collection will send a message to each instance (recursively) asking for its instance size and cardinality. It is implemented by a computed function.

# T_behavior

Supertypes: `T_object`
Subtypes: `T_algebra`
Native Behaviors:

**name**
*B_name* : `T_string`
Example: *B_name*(*o*)
Symbol:
Returns the signature name of the argument behavior *o*.

**argTypes**
*B_argTypes* : `T_list`⟨`T_type`⟩
Example: *B_argTypes*(*o*)
Symbol:
Returns the list of types that are the argument types of the signature for the behavior *o*.

**resultType**
*B_resultType* : `T_type`
Example: *B_resultType*(*o*)
Symbol:
Returns the type that is the result type of the signature for the behavior *o*.

**semantics**
*B_semantics* : `T_object`
Example: *B_semantics*(*o*)
Symbol: $[\![o]\!]$
Returns the full semantics of the argument behavior *o*.

**associate**
*B_associate* : `T_type` → `T_function` → `T_behavior`
Example: *B_associate*(*o*)(*p*)(*q*)
Symbol:
Associates the function object of the the third argument *q* with the behavior argument object *o* for the given type object *p*. The behavior has the side-effect of modifying the behavior *o* so that it executes the associated function *q* when applied to an object of type *p*.

**implementation**
*B_implementation* : `T_type` → `T_function`
Example: *B_implementation*(*o*)(*p*)
Symbol:
Returns the function object associated with the behavior argument object *o* for the argument type object *p*.

**primitiveApply**
*B_primitiveApply* : `T_object` → `T_object`
Example: *B_primitiveApply*(*o*)(*p*)
Symbol:
Applies the behavior object *o* to the argument object *p*. One of the requirements is that the type of *p* must define behavior *o* as part of its interface.

**defines**
*B_defines* : `T_collection`⟨`T_type`⟩
Example: *B_defines*(*o*)
Symbol:
Returns the collection of type objects that define the behavior argument object *o* as part of their interface.

Derived Behaviors:

**apply**

*B_apply* : `T_object` → `T_list` → `T_object`
Example: *B_apply*(*o*)(*p*)(*q*)
Symbol:
Derivation:
If the argument list *q* is null, the apply works the same as the primitive apply. If there are arguments, they are passed directly to the execution of the function associated with this behavior.

| Applies the behavior object *o* to the object *p* using the objects in the list *q* as arguments. The requirements are that the type of *p* must define behavior *o* as part of its interface and the type of the objects in *q* must conform to the arguments types defined by the signature of behavior *o*. |
|---|

Native Behaviors: Related to Query Optimization.

**costBehavior**

*B_costBehavior* : `T_type` → `T_list`⟨`T_type`⟩ → `T_real`
Example: *B_costBehavior*(*o*)(*p*)(*q*)
Symbol:

| It returns a pre-estimated cost of executing the behavior *o* on objects of type *p* with a list of arguments *q*. |
|---|

# T_function

Supertypes: T_object
Subtypes: T_query, T_searchStrat, T_context
Native Behaviors:

**name**
*B_name* : T_string
Example: *B_name*(*o*)
Symbol:
Returns the name of the function object *o*.

**argTypes**
*B_argTypes* : T_list⟨T_type⟩
Example: *B_argTypes*(*o*)
Symbol:
Returns a list of types which denote the types and ordering of the argument objects for the function argument object *o*.

**resultType**
*B_resultType* : T_type
Example: *B_resultType*(*o*)
Symbol:
Returns the result type of the function argument object *o*.

**comments**
*B_comments* : T_string
Example: *B_comments*(*o*)
Symbol:
Returns the comments that document the function object *o*.

**source**
*B_source* : T_object
Example: *B_source*(*o*)
Symbol:
Returns the source code of the function argument object *o*.

**primitiveExecute**
*B_primitiveExecute* : T_object → T_object
Example: *B_primitiveExecute*(*o*)(*p*)
Symbol:
Executes the function object *o* using the object *p* as an argument and returns a result object. One requirement is that the argument *p* must be compatible with the argument type of the function.

**compile**
*B_compile* : T_object
Example: *B_compile*(*o*)
Symbol:
Compiles the function argument object *o* and produces an executable which is returned by *B_executable* below.

**executable**
*B_executable* : T_object
Example: *B_executable*(*o*)
Symbol:
Returns the executable code of the function argument object *o*.

Derived Behaviors:

**execute**

*B_execute* : `T_list` → `T_object`
Example: *B_execute*(*o*)(*p*)
Symbol:
Derivation:
Function currying is abstracted as a list of arguments.

| Executes the function object *o* using the objects in the list *p* as arguments and returns a result object. One of the requirements is that the list of arguments in *p* must be compatible with the argument type list for the function. |
|---|

**basicExecute**

*B_basicExecute* : `T_list` → `T_object`
Example: *B_basicExecute*(*o*)(*p*)
Symbol:
Derivation:
Function currying is abstracted as a list of arguments.

| Executes the function object *o* using the objects in the list *p* as arguments and returns a result object. One of the requirements is that the list of arguments in *p* must be compatible with the argument type list for the function. |
|---|

Native Behaviors: Related to Query Optimization.

**costFunction**

*B_costFunction* : `T_list`⟨`T_object`⟩ → `T_costFunc`
Example: *B_costFunction*(*o*)(*p*)
Symbol:

| It returns a cost function object that when is executed returns a pre-estimated cost of executing the function behavior *o* with arguments *p*. It is implemented by a stored function. |
|---|

**cost**

*B_cost* : `T_list`⟨`T_object`⟩ → `T_integer`
Example: *B_cost*(*o*)(*p*)
Symbol:

| Executes the cost function object that is returned by *B_costFunction* and returns the estimated cost that results of this execution. It is implemented by a computed function. |
|---|

# T_query

Supertypes: T_function
Subtypes: T_adHoc, T_production
Overriden Behaviors: Replacement for Query Optimization purposes.

**name**
*B_name* : T_string
Example: *B_name*(*o*)
Symbol:

| Returns the name of the query object *o*. |
|---|

**argTypes**
*B_argTypes* : T_list⟨T_type⟩
Example: *B_argTypes*(*o*)
Symbol:

| Returns a list of types which denote the types and ordering of the argument objects for the query object *o*. The type of each of the elements of the list that is returned is either the object T_collection or any of its subtypes (i.e. T_class, T_bag). (It is implemented by a stored function) |
|---|

**resultType**
*B_resultType* : T_type
Example: *B_resultType*(*o*)
Symbol:

| Returns the result type of the execution of the query object *o*. The type that is returned is either the object T_collection or any of its subtypes , but T_class (i.e. T_bag). (It is implemented by a stored function) |
|---|

**comments**
*B_comments* : T_string
Example: *B_comments*(*o*)
Symbol:

| Returns the comments that document the query object *o*. |
|---|

**source**
*B_source* : T_string
Example: *B_source*(*o*)
Symbol:

| Returns the source code for a query *o* which is a TIGUKAT Query Language (TQL) statement. |
|---|

**executable** *B_executable* : `T_object`
Example: *B_executable*(*o*)
Symbol:

Returns the code that executes the Execution Plan for the optimized query object *o*. Following TIGUKAT Query Model, this Execution Plan which consists of the OAPT annotated with the algorithms that implement each algebraic node in the corresponding optimized OAPT, when the behavior *B_execute* is applied. Because algebraic nodes are functions, the source and executable code of the algorithms that implement them are stored in the *B_source* and *B_executable* behaviors respectively. In some cases, not only one Execution Plan is generated, but a family of Execution Plans; then, the Object Manager must choose the "best" Execution Plan based on cost estimations. The code that is contained in *B_executable* is the following: (OM.*B_chooseEP*(o.*B_execPlanFamily*())).*B_execute*()
Additional comments: when specializing `T_query` in ad-hoc and production, this behavior *B_executable* can have been optimized or not depending on the type of the query. If it is an ad-hoc query, then, it is possibly interpreted.

**basicExecute** *B_basicExecute* : `T_list`⟨`T_object`⟩ → `T_object`
Example: *B_basicExecute*(*o*)(*p*)
Symbol:

Submits the execution plan (or family of execution plans) object *p* for the query object *o* to the Object Manager for processing and returns the resulting collection object. In case of a family is passed to the OM, it must choose the "best" Execution Plan, before processing it. The code that is executed is (OM.*B_chooseEP*(o.*B_execPlanFamily*())).*B_execute*()

**basicExecSave** *B_basicExecSave* : `T_list`⟨`T_object`⟩ → `T_object`
Example: *B_basicExecSave*(*o*)(*p*)
Symbol:

Works the same as *B_basicExecute*. Aditionally, it saves the resulting collection in *B_result*.

**basicExecDontSave** *B_basicExecDontSave* : `T_list`⟨`T_object`⟩ → `T_object`
Example: *B_basicExecDontSave*(*o*)(*p*)
Symbol:

If *B_result* is **null**, it works the same as *B_basicExecute*. Otherwise, it checks the timestamps associated to the resulting collection stored in *B_result* and to the input collections to the query to decide whether to re-execute the query (applying *B_basicExecute* to the object *o*) or to return the collection stored in *B_result*. *B_basicExecDontSave* does not store the result in *B_result* in any case.

**execute**

*B_execute* : `T_list⟨T_object⟩` → `T_object`
Example: *B_execute*(*o*)(*p*)
Symbol:

| |
|---|
| Works the same as *B_basicExecDontSave*. Aditionally, it saves the new resulting collection in *B_result*, when the query *o* has been re-executed. |

**compile**

*B_compile* : `T_object`
Example: *B_compile*(*o*)
Symbol:

| |
|---|
| Compiles the source code for a query. The compilation process involves the following steps: translating the query statement *o* written in TQL language into an equivalent calculus expression; then, translating the calculus expression into an equivalent algebra expression and checking it for type consistency. In the next step, algebra optimization is performed by the behavior *B_optimize* that consists of applying equivalence preserving rewrite rules to the type consistent algebra expression. In the last step, the behavior *B_genExecPlan* generates an Execution Plan (or a family of Execution Plans) by annotating each individual algebra operator node from the optimized object algebra query processing tree with one of the algorithms that implement the corresponding node. These algorithms use object manager calls that are part of the low level object manipulation primitives that constitutes the interface to the Object Manager subsystem.<br>Future research must be done on picking the best algorithms that implement each algebraic node in the OAPT based on information provided by the OM such as indexes, clustering, and so on. As side effects of the application of the behavior *B_compile* on the query object *o*, the following behaviors are filled: *B_executable*, *B_initialOAPT*, *B_optimizedOAPT*, *B_transformations*, *B_argMbrTypes*, *B_rcvrMbrTypes*, *B_resultMbrType*, *B_execPlanFamily* and *B_result*. |

Native Behaviors: Related to Query Optimization.

**initialOAPT**

*B_initialOAPT* : `T_algOp`
Example: *B_initialOAPT*(*o*)
Symbol:

| |
|---|
| Returns the initial Object Algebra Processing Tree (OAPT) resulting from the calculus to algebra translation. This initial OAPT constitutes the initial state of the search space used for the algebraic optimization of the query object *o*. The initial OAPT must be complete for the optimization search strategy that we use. This is especially required when using randomized search strategies. This behavior is implemented by a stored function. |

**optimizedOAPT** *B_optimizedOAPT* : `T_list⟨T_algOp⟩`
Example: *B_optimizedOAPT*(*o*)
Symbol:

Returns the optimized OAPT (or list of optimized OAPTs) resulting from the optimization process for the query object *o*. This behavior is implemented by a stored function.

**searchStrat** *B_searchStrat* : `T_searchStrat`
Example: *B_searchStrat*(*o*)
Symbol:

Returns the search strategy that the optimizer (*B_optimize*) uses to control the optimization of the query object *o*. It must be determined externally, before *B_optimize* is applied (i.e. when the query object *o* is created). (It is implemented by a computed function).

**transformations** *B_transformations* : `T_list⟨T_algEqRule⟩`
Example: *B_transformations*(*o*)
Symbol:

Returns the list of transformation rule objects used for the algebraic optimization of the query object *o*. This behavior is implemented by a stored function.

**costModelFunc** *B_costModelFunc* : `T_costFunc`
Example: *B_costModelFunc*(*o*)
Symbol:

Returns the cost model function that the optimizer (*B_optimize*) uses when the search strategy (*B_searchStrat*) is a cost-controlled strategy. Otherwise, it returns **null**. It must be determined externally, before *B_optimize* is applied (i.e. when the query object *o* is created).

**argMbrTypes** *B_argMbrTypes* : `T_list⟨T_type⟩`
Example: *B_argMbrTypes*(*o*)
Symbol:

Returns a list whose elements corresponds to the membership type object for each of the target collections to the query object *o*.

**resultMbrType** *B_resultMbrType* : `T_type`
Example: *B_resultMbrType*(*o*)
Symbol:

Returns the membership type object of the resulting collection from executing the query object *o*.

**optimize** *B_optimize* : `T_list⟨T_algOp⟩`
Example: *B_optimize*(*o*)
Symbol:

It starts the execution of the algebraic query optimizer over the query object *o*, using the search strategy object *o.B_searchStrat*, and taking the initial OAPT object *o.B_initialOAPT* as the initial state of the search space. This behavior will be invoked by the behavior *B_compile*.

**genExecPlan**

*B_genExecPlan* : T_collection⟨T_function⟩
Example: *B_genExecPlan*(*o*)
Symbol:

Generates an Execution Plan (or a family of execution plans) from the optimized OAPT object *o.B_optimizedOAPT* for the query object *o*. The Execution Plan is modeled as a T_algOp object that executes the query object *o*. The Execution Plan (or a family of Execution Plans) is created by annotating each individual algebra operator from the optimized processing tree (OAPT) with the algorithm that implements it. These algorithms use object manager calls that are part of the low level object manipulation primitives that constitutes the interface to the Object Manager subsystem.
Since each node of the OAPT might be implemented by different algorithms, a collection of OAPTs might be the result of applying the behavior *B_genExecPlan*. This behavior is invoked by the behavior *B_compile*.
As a side effect, *B_genExecPlan* stores the resulting Execution Plan into *B_execPlanFamily* behavior.

**execPlanFamily**

*B_execPlanFamily* : T_collection⟨T_function⟩
Example: *B_execPlanFamily*(*o*)
Symbol:

Returns an Execution Plan (or a family of execution plans) that are generated by *B_genExecPlan*.

**budgetOpt**

*B_budgetOpt* : T_integer
Example: *B_budgetOpt*(*o*)
Symbol:

Returns the optimization budget that has been assigned to the query object *o*. *B_budgetOpt* provides an upper bound for optimization cost which can be used by the search strategy that controls the optimization of the query *o*. (It is implemented by a stored function). A value is assigned to this behavior by the "user" or the system (i.e., when the query object is created).

**lastOpt**

*B_lastOpt* : T_date
Example: *B_lastOpt*(*o*)
Symbol:

Returns the last date in that the query object *o* was optimized. It can be useful for checking consistency between an optimized query and the characteristics of the target collections that were used for its optimization (i.e., variations in the cardinality of one of the input collections).

**lastExec**

*B_lastExec* : `T_date`
Example: *B_lastExec*(*o*)
Symbol:

| |
|---|
| Returns the last date in that the query object *o* was executed. It can be useful for checking consistency between the result stored in *B_result* and changes in the extensions of the target collections to the query. |

Native Behaviors:

**result**

*B_result* : `T_collection`
Example: *B_result*(*o*)
Symbol:

| |
|---|
| Returns the query result that was stored in *B_result* after the query *o* was executed indicating to save the result collection object by applying either *B_basicExecSave* or *B_execute* to the object *o*. Otherwise, it returns **null**. |

# T_adHoc

Supertypes: `T_query`
Subtypes: *none*
Overriden Behaviors: Replacement for Query Optimization purposes.

**source** *B_source* : `T_object`
Example: *B_source*(*o*)
Symbol:

| It returns the source code for a query *o* which is a TIGUKAT Query Language (TQL) statement. |
|---|

**executable** *B_executable* : `T_object`
Example: *B_executable*(*o*)
Symbol:

| It returns the executable code for a query *o*. This code is possibly interpreted. It might have not been optimized because it is an ad-hoc query. |
|---|

**execute** *B_execute* : `T_list`⟨`T_object`⟩ → `T_object`
Example: *B_execute*(*o*)(*p*)
Symbol:

| It interprets the code. |
|---|

**optimize** *B_optimize* : `T_list`⟨`T_algOp`⟩
Example: *B_optimize*(*o*)(*p*)(*q*)
Symbol:

| It starts the execution of the algebraic query optimizer over the query object *o*, This behavior will be invoked by the behavior *B_compile*. |
|---|

# T_production

Supertypes: T_query
Subtypes: *none*
Overriden Behaviors: Replacement for Query Optimization purposes.

**source**
*B_source* : T_object
Example: *B_source*(*o*)
Symbol:

It returns the source code for a query *o* which is a TIGUKAT Query Language (TQL) statement.

**executable**
*B_executable* : T_object
Example: *B_executable*(*o*)
Symbol:

It returns the executable code for a production query object *o* which has been compiled. This code will be optimized once and then executed many times. For this reason, the optimization process of these queries might incurre in high compile-time optimization strategies.

**execute**
*B_execute* : T_list⟨T_object⟩ → T_object
Example: *B_execute*(*o*)(*p*)
Symbol:

It executes the compiled code.

**optimize**
*B_optimize* : T_list⟨T_algOp⟩
Example: *B_optimize*(*o*)
Symbol:

It starts the execution of the algebraic query optimizer to optimize the query object *o*. This behavior will be used by the behavior *B_compile*.

# T_context

Supertypes: T_function
Subtypes: T_algOp
Overriden Behaviors: Replacement for Query Optimization purposes.

**name**
*B_name* : T_string
Example: *B_name*(*o*)
Symbol:

| Returns the signature name of the particular instance *o*. (It is implemented by a stored function) |
|---|

**argTypes**
*B_argTypes* : T_list⟨T_type⟩
Example: *B_argTypes*(*o*)
Symbol:

| Returns the list of types corresponding to the argument types of the signature for the behavior object that the context node object *o* is representing. (It is implemented by a stored function) |
|---|

**resultType**
*B_resultType* : T_type
Example: *B_resultType*(*o*)
Symbol:

| Returns the type of the result of applying the behavior object that the context node object *o* is representing. (It is implemented by a stored function) |
|---|

**comments**
*B_comments* : T_string
Example: *B_comments*(*o*)
Symbol:

| An object of type T_context models the delayed execution of a behavior object which is possible part of a composition of behaviors, also called a path expression. We name this objects as context nodes. |
|---|

**source**
*B_source* : T_string
Example: *B_source*(*o*)
Symbol:

| Returns the source code of the function object *o*. |
|---|

**execute**
*B_execute* : T_list⟨T_object⟩ → T_object
Example: *B_execute*(*o*)(*p*)
Symbol:

| Executes the function object *o* using the object *p* as an argument, and returns a collection object. This collection results from executing recursively the context tree for which the node *o* is the root. The type of *p* must be compatible with the argument type of the function. |
|---|

**basicExecute**
*B_basicExecute* : T_list⟨T_object⟩ → T_object
Example: *B_basicExecute*(*o*)(*p*)
Symbol:

| Works the same as *B_execute* that is defined above. |
|---|

**compile** *B_compile* : `T_object`
Example: *B_compile*(*o*)
Symbol:

| |
|---|
| Compiles the function argument object *o* and produces an executable which is returned by *B_executable* below. |

**executable** *B_executable* : `T_object`
Example: *B_executable*(*o*)
Symbol:

| |
|---|
| Returns the executable code for the context node *o*. |

**costFunction** *B_costFunction* : `T_list`⟨`T_object`⟩ → `T_function`
Example: *B_costFunction*(*o*)(*p*)
Symbol:

| |
|---|
| Returns a cost function object that when is executed returns a pre-estimated cost of executing the function *o* with arguments *p*. The computation of the cost of a context node object includes calculating recursively the cost of its children plus the cost of its own execution. It is implemented by a computed function. |

**cost** *B_cost* : `T_list`⟨`T_object`⟩ → `T_integer`
Example: *B_cost*(*o*)(*p*)
Symbol:

| |
|---|
| Executes the cost function object that is returned by *B_costFunction* and returns the estimated cost that results of this execution. It is implemented by a computed function. |

Native Behaviors: Related to Query Optimization.

**outerRcvr** *B_outerRcvr* : `T_context`
Example: *B_outerRcvr*(*o*)
Symbol:

| |
|---|
| Returns a reference to the context node object of type `T_context` that has the role of receiver object of the action of applying on it the behavior (i.e. *B_select*) that the node object *o* is representing. The behavior *B_outerRcvr* might return a reference to a context subtree. |

**innerArg** *B_innerArg* : `T_list`⟨`T_context`⟩
Example: *B_innerArg*(*o*)
Symbol:

| |
|---|
| Returns a list of references to the context node objects of type `T_context` that correspond to the arguments for the behavior whose delayed execution is represented by the node object *o*. Each of the arguments might reference a context subtree. |

**rcvrType** *B_rcvrType* : `T_type`
Example: *B_rcvrType*(*o*)
Symbol:

| |
|---|
| Returns the type of the receiver object for the function object that the context node object *o* is representing. (It is implemented by a stored function) |

# T_algOp

Supertypes: `T_context`

Instances: *F_leaf*, *F_select*, *F_generate*, *F_map*, *F_project*, *F_difference*, *F_union*, *F_intersection*, *F_product*, *F_join*

Overriden Behaviors: Replacement for Query Optimization purposes.

**name**

*B_name* : `T_string`
Example: *B_name*(*o*)
Symbol:

| Returns the signature name of the particular instance *o*. (It is implemented by a stored function) |
|---|

**rcvrType**

*B_rcvrType* : `T_type`
Example: *B_rcvrType*(*o*)
Symbol:

| Returns the type of the receiver object for the algebraic operation behavior that the node object *o* is representing. The type that is returned is either the object `T_collection` or any of its subtypes (i.e. `T_class`, `T_bag`). (It is implemented by a stored function) |
|---|

**argTypes**

*B_argTypes* : `T_list`⟨`T_type`⟩
Example: *B_argTypes*(*o*)
Symbol:

| Returns the list of types corresponding to the argument types of the signature for the algebraic operator that the node object *o* is representing. The types that are returned are either the object `T_collection` or any of its subtypes (i.e. `T_class`, `T_bag`). (It is implemented by a stored function) |
|---|

**resultType**

*B_resultType* : `T_type`
Example: *B_resultType*(*o*)
Symbol:

| Returns the type object `T_collection` (or any of its subtypes, but `T_class`) that is the type of the result of applying the algebraic operator that the node object *o* is representing because of the closure of the algebra. (It is implemented by a stored function) |
|---|

**comments**

*B_comments* : `T_string`
Example: *B_comments*(*o*)
Symbol:

| The type `T_algOp` is the type of the node objects of an OAPT which models the delayed execution of a behavioral composition of algebraic operators. These algebraic operators are defined as behaviors in the interface of the type `T_collection` (i.e. *B_select*). |
|---|

**source**

*B_source* : `T_string`
Example: *B_source*(*o*)
Symbol:

| Returns the source code of the function object *o*. |
|---|

**execute**

*B_execute* : T_list⟨T_collection⟩ → T_collection
Example: *B_execute*(*o*)(*p*)
Symbol:

| |
|---|
| Executes the function object *o* using the objects in the list *p* as arguments and returns a collection object. This collection results from executing recursively the OAPT for which the node *o* is the root. As a side effect, the resulting collection is stored in *B_targetColl*. One of the requirements is that the list of arguments in *p* must be compatible with the argument type list for the function. |

**basicExecute**

*B_basicExecute* : T_list⟨T_collection⟩ → T_collection
Example: *B_basicExecute*(*o*)(*p*)
Symbol:

| |
|---|
| It works the same as *B_execute*. |

**compile**

*B_compile* : T_object
Example: *B_compile*(*o*)
Symbol:

| |
|---|
| Compiles the function argument object *o* and produces an executable which is returned by *B_executable* below. |

**executable**

*B_executable* : T_object
Example: *B_executable*(*o*)
Symbol:

| |
|---|
| Returns the code that executes the algorithm that implements the algebraic operator that the node *o* is modeling. |

**costFunction**

*B_costFunction* : T_list⟨T_object⟩ → T_function
Example: *B_costFunction*(*o*)(*p*)
Symbol:

| |
|---|
| Returns a cost function object that when is executed returns a pre-estimated cost of executing the algebraic node *o* with arguments *p*. It is implemented by a stored function. |

**cost**

*B_cost* : T_list⟨T_object⟩ → T_integer
Example: *B_cost*(*o*)(*p*)
Symbol:

| |
|---|
| Executes the cost function object that is returned by *B_costFunction* and returns the estimated cost that results of this execution. It is implemented by a computed function. |

**outerRcvr**

*B_outerRcvr* : T_algOp
Example: *B_outerRcvr*(*o*)
Symbol:

| |
|---|
| Returns a reference to the algebraic operator node object of type T_algOp that has the role of receiver object of the action of applying on it the algebraic operation behavior (i.e. *B_select*) that the node object *o* is representing. The behavior *B_outerRcvr* might return a reference to an OAP subtree. |

**innerArgs** *B_innerArgs* : `T_list⟨T_algOp⟩`
Example: *B_innerArgs*(*o*)
Symbol:

| Returns a list of references to the algebraic operator node objects of type `T_algOp` that correspond to the arguments for the algebraic operator that is represented by the node object *o*. Each of the arguments might reference an OAP subtree. |
|---|

Native Behaviors: Related to Query Optimization.

**rcvrMbrType** *B_rcvrMbrType* : `T_type`
Example: *B_rcvrMbrType*(*o*)
Symbol:

| Returns the membership type object of the receiver collection object for the algebraic operation behavior that the node object *o* is representing. |
|---|

**argMbrTypes** *B_argMbrTypes* : `T_list⟨T_type⟩`
Example: *B_argMbrTypes*(*o*)
Symbol:

| Returns a list whose elements are the membership type objects corresponding to the argument collection objects for the algebraic operator that the node object *o* is representing. |
|---|

**resultMbrType** *B_resultMbrType* : `T_type`
Example: *B_resultMbrType*(*o*)
Symbol:

| Returns the membership type object of the resulting collection from executing the algebraic operation behavior that the node object *o* is representing. |
|---|

**targetVar** *B_targetVar* : `T_var`
Example: *B_targetVar*(*o*)
Symbol:

| Returns a reference to the target variable object for the algebraic operator that the node *o* is representing. |
|---|

**targetColl** *B_targetColl* : `T_collection`
Example: *B_targetColl*(*o*)
Symbol:

| Returns a reference to the target collection object that results from executing the algebraic operator that the node *o* is modeling. |
|---|

**constraint** *B_constraint* : `T_object`
Example: *B_constraint*(*o*)
Symbol:

| Returns an object that models a constraint on the algebraic operator that the context node object *o* is representing. For example, a formula that qualifies the select operator, or the list of behaviors that must be applied to the receiver and argument collections of the map operator are constraints on the select and map operators respectively. |
|---|

**execAlgorithm** *B_execAlgorithm* : `T_function`
Example: *B_execAlgorithm*(*o*)
Symbol:

| |
|---|
| Returns a function object that implements an execution algorithm for the algebraic operation that the node *o* represents. It is implemented by a stored function. |

Native Behaviors: Operations on trees.

**splitLeft** *B_splitLeft* : `T_algOp`
Example: *B_splitLeft*(*o*)
Symbol:

| |
|---|
| Returns the subtree that corresponds to the *B_outerRcvr* of the OAPT *o*. As a side effect, it sets to **null** the behavior *B_outerRcvr* for the node *o*. |

**splitRight** *B_splitRight* : `T_list`⟨`T_algOp`⟩
Example: *B_splitRight*(*o*)
Symbol:

| |
|---|
| Returns the list of OAPTs that corresponds to the *B_innerArgs* of the OAPT *o*. As a side effect, it sets to **null** the behavior *B_innerArgs* for the node *o*. |

**linkLeft** *B_linkLeft* : `T_algOp` → `T_algOp`
Example: *B_linkLeft*(*o*)(*p*)
Symbol:

| |
|---|
| Links the OAPT *p* to the node *o* as *o*'s left son *B_outerRcvr*. Returns the node *o*. |

**linkRight** *B_linkRight* : `T_list`⟨`T_algOp`⟩ → `T_algOp`
Example: *B_linkRight*(*o*)(*p*)
Symbol:

| |
|---|
| Links the list of OAPTs *p* to the node *o* as *o*'s right son *B_innerArgs*. Returns the node *o*. |

**assemble** *B_assemble* : `T_algOp` → `T_list`⟨`T_algOp`⟩ → `T_algOp`
Example: *B_assemble*(*o*)(*p*)(*q*)
Symbol:

| |
|---|
| Given the OAPT *p*, the list of OAPTs *q* and the node *o*, *B_assemble* combines them into a single OAPT with root *o*, left son (*B_outerRcvr*) *p*, and right son (*B_innerArgs*) *q*. Returns the OAPT rooted at *o*. |

**dissassemble** *B_dissassemble* : `T_list`⟨`T_object`⟩
Example: *B_assemble*(*o*)
Symbol:

| |
|---|
| Breaks the OAPT rooted at *o* into three parts: an OAPT containing only the node *o*, and the left and right children of *o*. Returns a list containing at most two elements: the first element is the left son (*B_outerRcvr*), and the second element is the right son (*B_innerArgs*). As a side effect, it sets to **null** the behaviors *B_outerRcvr* and *B_innerArgs* for the OAPT *o*. |

# T_formula

Supertypes: T_function
Subtypes: T_atom

**name**
*B_name* : T_string
Example: *B_name*(*o*)
Symbol:

| Returns the name of the function object *o*. |
|---|

**argTypes**
*B_argTypes* : T_null
Example: *B_argTypes*(*o*)
Symbol:

| Returns the object null because this function does not have any arguments. |
|---|

**resultType**
*B_resultType* : T_type
Example: *B_resultType*(*o*)
Symbol:

| Returns the type T_boolean that is the result type of the function object *o*. |
|---|

**source**
*B_source* : T_string
Example: *B_source*(*o*)
Symbol:

| Returns the expression that specifies the formula object *o*. |
|---|

**compile**
*B_compile* : T_context
Example: *B_compile*(*o*)
Symbol:

| Compiles the source code of the atom object *o* and produces its delayed execution object which is returned by *B_executable* below. |
|---|

**executable**
*B_executable* : T_context
Example: *B_executable*(*o*)
Symbol:

| Returns the delayed execution object, a context object, for the expression that specifies the atom object *o*. When the context object is executed, it returns a boolean object. |
|---|

**execute**
*B_execute* : T_null → T_boolean
Example: *B_execute*(*o*)(*p*)
Symbol:

| Executes the context object that is stored in *B_executable*. The execution of the context node returns a boolean object. The context object models the delayed execution of the atom *o*. The code that is executed is (o.*B_executable*()).*B_execute*. |
|---|

**basicExecute**
*B_basicExecute* : T_null → T_boolean
Example: *B_basicExecute*(*o*)(*p*)
Symbol:

| It works the same as *B_execute*. |
|---|

Native Behaviors:

**atoms**

*B_atoms* : T_list⟨T_atom⟩
Example: *B_atoms*(*o*)
Symbol:

| Returns the list of atoms that are referenced in the formula object *o*. It is implemented by a stored function. |
|---|

**CNF**

*B_CNF* : T_list⟨T_list⟨T_atom⟩⟩
Example: *B_CNF*(*o*)
Symbol:

| Returns a representation for the formula, which is given in Conjunctive Normal Form. This representation is interpreted in the following way: the elements of the outer list, which are list of atoms, are connected by conjunctions; the elements of the inner list, which are atoms, are connected by disjunctions. Assumption: atoms are assumed to be positive. |
|---|

**restVar**

*B_restVar* : T_list⟨T_var⟩
Example: *B_restVar*(*o*)
Symbol:

| Returns the list of restricted variables in the atoms that are referenced by the formula object *o*. Restricted variables are variables that are not generated by any atom in the formula *o*. It is implemented by a computed function. |
|---|

**genVar**

*B_genVar* : T_var
Example: *B_genVar*(*o*)
Symbol:

| Returns the variable that is generated by one of more atoms in the formula object *o*. A formula can have only one generated variable because of constraints in the object algebra. It is implemented by a computed function. |
|---|

**refVar**

*B_refVar* : T_list⟨T_var⟩
Example: *B_refVar*(*o*)
Symbol:

| Returns the list of variables that are referenced in the atoms of the formula *o*. It is implemented by a computed function that performs the union between the list *B_restVar* and the list containing the generated variable *B_genVar*. |
|---|

**splitRestrDisj**

*B_splitRestrDisj* : T_collection
Example: *B_splitRestrDisj*(*o*)(*p*)
Symbol:

| Returns the collection of formulas in CNF form that result from splitting the formula *o* into *p* number of parts which were connected by disjunctions. |
|---|

**splitRestrConj** *B_splitRestrConj* : `T_collection`
Example: *B_splitRestrConj*(*o*)(*p*)
Symbol:

Returns the collection of formulas in CNF form that result from splitting the formula *o* into *p* number of parts which were connected by conjunctions.

# T_atom

Supertypes: T_formula
Subtypes: *none*
Overriden Behaviors: Replacement for Query Optimization purposes.

**name**
*B_name* : T_string
Example: *B_name*(*o*)
Symbol:

| Returns the name of the function object *o*. |
|---|

**argTypes**
*B_argTypes* : T_null
Example: *B_argTypes*(*o*)
Symbol:

| Returns the object **null** because this function does not have any arguments. |
|---|

**resultType**
*B_resultType* : T_type
Example: *B_resultType*(*o*)
Symbol:

| Returns the type T_boolean that is the result type of the function object *o*. |
|---|

**source**
*B_source* : T_string
Example: *B_source*(*o*)
Symbol:

| Returns the expression that specifies the atom object *o*. |
|---|

**compile**
*B_compile* : T_context
Example: *B_compile*(*o*)
Symbol:

| Compiles the source code of the atom object *o* and produces its delayed execution object which is returned by *B_executable* below.. |
|---|

**executable**
*B_executable* : T_context
Example: *B_executable*(*o*)
Symbol:

| Returns the delayed execution object, a context object, for the expression that specifies the atom object *o*. When the context object is executed, it returns a boolean object. |
|---|

**execute**
*B_execute* : T_null → T_boolean
Example: *B_execute*(*o*)(*p*)
Symbol:

| Executes the context object that is stored in *B_executable*. The execution of the context node returns a boolean object. The context object models the delayed execution of the atom *o*. The code that is executed is (o.*B_executable*()).*B_execute*. |
|---|

**basicExecute**
*B_basicExecute* : T_null → T_boolean
Example: *B_basicExecute*(*o*)(*p*)
Symbol:

| It works the same as *B_execute*. |
|---|

**atoms**

*B_atoms* : T_list⟨T_atom⟩
Example: *B_atoms*(*o*)
Symbol:

| |
|---|
| Returns a list containing only one element that is the atom object *o*. |

**CNF**

*B_CNF* : T_null
Example: *B_CNF*(*o*)
Symbol:

| |
|---|
| Returns the object **null** because an atom is the minimal building block for formulas. |

**restVar**

*B_restVar* : T_list⟨T_var⟩
Example: *B_restVar*(*o*)
Symbol:

| |
|---|
| Returns the list of restricted variables in the atom *o*. Restricted variables are variables that are not generated by the atom *o*. It is implemented by a stored function. |

**genVar**

*B_genVar* : T_var
Example: *B_genVar*(*o*)
Symbol:

| |
|---|
| Returns the variable that is generated by the atom *o*. An atom can only generate one variable. It is implemented by a stored function. |

**refVar**

*B_refVar* : T_list⟨T_var⟩
Example: *B_refVar*(*o*)
Symbol:

| |
|---|
| Returns the list of variables that are referenced in the atom *o*. It is implemented by a computed function that performs the union between the list *B_restVar* and the list containing the generated variable *B_genVar*. |

# T_var

Supertypes: T_object
Subtypes: *none*
Native Behaviors: Related to Query Optimization.

**atomRef** *B_atomRef* : T_list⟨T_atom⟩
Example: *B_atomsRef*(*o*)
Symbol:

| Returns the list of atoms that reference the variable object *o*. |
|---|

**algOpRef** *B_algOpRef* : T_algOp
Example: *B_algOpRef*(*o*)
Symbol:

| Returns the algebraic node that has the variable object *o* as its target variable *B_targetVar*. |
|---|

# T_rule

Supertypes: T_object
Subtypes: T_algEqRule
Native Behaviors: Related to Query Optimization.

**cond** *B_cond* : T_list⟨T_formula⟩
Example: *B_cond*(*o*)
Symbol:

Returns an object that models the condition that must be satisfied by the object *p* in order to apply the rule *o*.

**checkCond** *B_checkCond* : T_object → T_boolean
Example: *B_checkCond*(*o*)(*p*)
Symbol:

Checks if the condition stored in *B_cond* holds for the object *p*. If so, the object **true** is returned. Otherwise, **false** is returned.

**action** *B_action* : T_object → T_object
Example: *B_action*(*o*)(*p*)
Symbol:

Returns the object resulting from applying the action dictated by the rule object *o* to the argument object *p*.

# T_algEqRule

Supertypes: T_rule
Subtypes: *none*
Overriden Behaviors: Replacement for Query Optimization purposes.

**cond**
*B_cond* : T_list⟨T_formula⟩
Example: *B_cond*(*o*)
Symbol:

| Returns a function object that implements for the condition associated to the right side of the rule object *o*. (Inherited from T_rule, but overloaded). |
|---|

**checkCond**
*B_checkCond* : T_algOp → T_boolean
Example: *B_checkCond*(*o*)(*p*)
Symbol:

| Executes the function object stored in *B_cond*. If the argument object *p* holds the condition associated to the right side of the rule object *o*, the object **true** is returned. Otherwise, **false** is returned. (Inherited from T_rule, but overloaded). It is implemented by a computed function. |
|---|

**action**
*B_action* : T_algOp → T_algOp
Example: *B_action*(*o*)(*p*)
Symbol:

| Executes the function object stored in *B_actionRightFunc* passing *p* as argument. It returns the OAPT object resulting from applying the transformation dictated by the rule object *o* to the argument object *p*. The resulting OAPT have the shape of the expression given in the left side of the rule. (Inherited from T_rule, but overloaded). |
|---|

Native Behaviors: Related to Query Optimization.

**leftSideFunc**
*B_leftSideFunc* : T_algOp → T_function
Example: *B_leftSide*(*o*)(*p*)
Symbol:

| Returns the function object that implements the matching algorithm that corresponds to the left side expression of the rule object *o*. It is implemented by a stored function. |
|---|

**matchLeft**
*B_matchLeft* : T_algOp → T_boolean
Example: *B_matchLeft*(*o*)(*p*)
Symbol:

| Executes the function object stored in *B_leftSideFunc* passing *p* as argument. If the argument object *p* matches the left side of the algebraic equivalence rule, the object **true** is returned. Otherwise **false** is returned. It is implemented by a computed function. |
|---|

**rightSideFunc** *B_rightSideFunc* : `T_algOp` → `T_function`
Example: *B_rightSideFunc*(*o*)(*p*)
Symbol:

Returns the function object that implements the matching algorithm that corresponds to the right side expression of the rule object *o*. It is implemented by a stored function.

**matchRight** *B_matchRight* : `T_algOp` → `T_boolean`
Example: *B_matchRigth*(*o*)(*p*)
Symbol:

Executes the function object stored in *B_rightSideFunc* passing *p* as argument. If the argument object *p* matches the right side of the algebraic equivalence rule, the object **true** is returned. Otherwise **false** is returned. It is implemented by a computed function.

**condLeft** *B_condLeft* : `T_list`(`T_formula`)
Example: *B_condLeft*(*o*)
Symbol:

Returns a function object that implements the condition associated to the left side expression of the rule object *o*. It is implemented by a stored function.

**checkCondLeft** *B_checkCondLeft* : `T_algOp` → `T_boolean`
Example: *B_checkCondLeft*(*o*)(*p*)
Symbol:

Executes the function object stored in *B_condLeft*. If the argument object *p* holds the condition associated to the left side of the rule object *o*, the object **true** is returned. Otherwise, **false** is returned. It is implemented by a computed function.

**actionLeftFunc** *B_actionLeftFunc* : `T_algOp` → `T_function`
Example: *B_actionLeft*(*o*)(*p*)
Symbol:

Returns the function object that implements the transformation dictated by the left side of the rule object *o* to the argument object *p*. It is implemented by a stored function.

**actionLeft** *B_actionLeft* : `T_algOp` → `T_algOp`
Example: *B_actionLeft*(*o*)(*p*)
Symbol:

Executes the function object stored in *B_leftSideFunc* passing *p* as argument. It returns the OAPT object resulting from applying the transformation dictated by the rule object *o* to the argument object *p*. The resulting OAPT have the shape of the algebraic expression for the right side of the rule object *o*. It is implemented by a computed function.

**actionRightFunc** *B_actionRightFunc* : `T_algOp` → `T_function`
Example: *B_actionRightFunc*(*o*)(*p*)
Symbol:

Returns the function object that implements the transformation dictated by the right side of the rule object *o* to the argument object *p*. It is implemented by a stored function.

**algExpression** *B_algExpression* : `T_object`
Example: *B_algExpression*(*o*)
Symbol:

| Returns the algebraic expression that specifies the rule object *o*. |
|---|

# T_searchStrat

Supertypes: T_function
Subtypes: T_enumSS, T_randomSS, T_heurSS
Overriden Behaviors: Replacement for Query Optimization purposes.

**argTypes**
*B_argTypes* : T_list⟨T_type⟩
Example: *B_argTypes*(*o*)
Symbol:

| Returns a list of types which denote the types and ordering of the argument objects for the search strategy object *o*. |
|---|

**resultType**
*B_resultType* : T_type
Example: *B_resultType*(*o*)
Symbol:

| Returns the result type of the search strategy object *o*. |
|---|

**source**
*B_source* : T_object
Example: *B_source*(*o*)
Symbol:

| Returns the source code of the search strategy object *o*. |
|---|

**compile**
*B_compile* : T_object
Example: *B_compile*(*o*)
Symbol:

| Compiles the search strategy object *o* and produces executable code which is stored in *B_executable*. |
|---|

**executable**
*B_executable* : T_object
Example: *B_executable*(*o*)
Symbol:

| Returns the executable code of the search strategy object *o*. |
|---|

**execute**
*B_execute* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_execute*(*o*)(*p*)
Symbol:

| Executes the search strategy object *o* using the list of OAPT objects *p* as arguments and returns an "optimal" OAPT object. One requirement is that the list of arguments *p* must be compatible with the argument type list for the function. |
|---|

**basicExecute**
*B_basicExecute* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_basicExecute*(*o*)(*p*)
Symbol:

| It works the same as *B_execute*. |
|---|

**costFunction**
*B_costFunction* : T_list⟨T_object⟩ → T_function
Example: *B_costFunction*(*o*)(*p*)
Symbol:

| It returns a cost function object that when is executed returns a pre-estimated cost of executing the search strategy object *o* with arguments *p*. This information is useful when choosing the proper search strategy for the algebraic optimization of a query. |
|---|

Native Behaviors: Related to Query Optimization.

**goal**

*B_goal* : `T_collection⟨T_algOp⟩`
Example: *B_goal*(*o*)
Symbol:

| Returns the collection of states that have been chosen as "good" candidates for being returned as the optimized OAPT. |
|---|

**optimal**

*B_optimal* : `T_algOp`
Example: *B_optimal*(*o*)
Symbol:

| Returns the "optimal" OAPT from the collection *B_goal*. |
|---|

**initSS**

*B_initSS* : `T_object`
Example: *B_initSS*(*o*)
Symbol:

| Returns the initial state(s) of the search space from where the search strategy object *o* starts the search. The result of applying this behavior are the `T_algOp` objects passed as arguments to the search strategy. |
|---|

**stopCond**

*B_stopCond* : `T_boolean`
Example: *B_stopCond*(*o*)
Symbol:

| Returns the object **true** if the condition given to stop the search process *o* holds. Otherwise **false** is returned. |
|---|

**setNextState**

*B_setNextState* : `T_list⟨T_algOp⟩` → `T_algOp`
Example: *B_setNextState*(*o*)(*p*)
Symbol:

| Returns the next state in the search space to be applied and action on. It determines in which way the states are investigated in the search space. If its implementation chooses the least recent state, then the search strategy is breadth-first; if it chooses the most recently generated state, then it implements depth-first search. |
|---|

**action**

*B_action* : `T_algOp` → `T_list⟨T_algOp⟩`
Example: *B_action*(*o*)(*p*)
Symbol:

| Generates a list of successor states for the state *p* by applying algebraic equivalence transformation rules on it. |
|---|

# T_heurSS

Supertypes: T_searchStrat
Subtypes: T_CCHeurSS
Overriden Behaviors: Replacement for Query Optimization purposes.

**initSS**
*B_initSS* : T_algOp
Example: *B_initSS*(*o*)
Symbol:

Returns the initial OAPT of the search space from where the search strategy object *o* will start the search. As a side effect, it initializes *B_current* (*B_setCurrent*) and *B_transfRules* (*B_setTransRules*).

**stopCond**
*B_stopCond* : T_boolean
Example: *B_stopCond*(*o*)
Symbol:

Returns the object **true** if the condition given to stop the search process *o* holds. Otherwise **false** is returned.

**setNextState**
*B_setNextState* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_setNextState*(*o*)(*p*)
Symbol:

Returns the next state in the search space to be applied and action on. It determines in which way the states are investigated in the search space.

**action**
*B_action* : T_algOp → T_list⟨T_algOp⟩
Example: *B_action*(*o*)(*p*)
Symbol:

Generates a list of successor states for the state *p* by applying algebraic equivalence transformation rules on *p* that are chosen according to heuristics defined for the search.

**argTypes**
*B_argTypes* : T_list⟨T_type⟩
Example: *B_argTypes*(*o*)
Symbol:

Returns a list of types which denote the types and ordering of the argument objects for the heuristic search strategy object *o*.

**resultType**
*B_resultType* : T_type
Example: *B_resultType*(*o*)
Symbol:

Returns the result type of the search strategy object *o*.

**source**
*B_source* : T_object
Example: *B_source*(*o*)
Symbol:

Returns the source code of the heuristic search strategy object *o*.

**compile**
*B_compile* : T_object
Example: *B_compile*(*o*)
Symbol:

Compiles the heuristic search strategy object *o* and produces executable code which is stored in *B_executable*.

**executable** *B_executable* : T_object
Example: *B_executable*(*o*)
Symbol:

| |
|---|
| Returns the executable code of the heuristic search strategy object *o*. |

**execute** *B_execute* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_execute*(*o*)(*p*)
Symbol:

| |
|---|
| Executes the heuristic search strategy object *o* using the list of OAPT objects *p* as arguments and returns an "optimal" OAPT object. One requirement is that the list of arguments *p* must be compatible with the argument type list for the function. |

**basicExecute** *B_basicExecute* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_basicExecute*(*o*)(*p*)
Symbol:

| |
|---|
| It works the same as *B_execute*. |

**costFunction** *B_costFunction* : T_list⟨T_object⟩ → T_function
Example: *B_costFunction*(*o*)(*p*)
Symbol:

| |
|---|
| It returns a cost function object that when is executed returns a pre-estimated cost of executing the heuristic search strategy object *o* with arguments *p*. |

Native Behaviors: Related to Query Optimization.

**current** *B_current* : T_algOp
Example: *B_current*(*o*)
Symbol:

| |
|---|
| Returns the current state of the search space that is being explored by the search strategy *o*. It is generated by the behavior *B_setNextState*. (It is implemented by a stored function). |

**acceptAction** *B_acceptAction* : T_algOp → T_boolean
Example: *B_acceptAction*(*o*)(*p*)
Symbol:

| |
|---|
| Returns **true** if the OAPT *p* meets the criteria as defined for the heuristic search strategy. (i.e. if the OAPT is a bushy or a linear tree). Otherwise, it returns **false**. |

**transfRules** *B_transfRules* : T_list
Example: *B_transfRules*(*o*)
Symbol:

| |
|---|
| Returns the list of transformation rules that are applied by the search strategy *o*. It must be set when the search strategy object *o* is first created. The list is ordered by priority of the rule. (It is implemented by a stored function). |

**chooseRule**

*B_chooseRule* : `T_algEqRule`
Example: *B_chooseRule*(*o*)
Symbol:

Returns the current element in the list *B_transfRules*. The next element becomes the current one. (It is implemented by a computed function).

# T_CCHeurSS

Supertypes: T_heurSS
Subtypes: *none*
Overriden Behaviors: Replacement for Query Optimization purposes.

**current**

*B_current* : T_algOp
Example: *B_current*(*o*)
Symbol:

Returns the current state of the search space that is being explored by the search strategy *o*. It is generated by the behavior *B_setNextState*. (It is implemented by a stored function).

**acceptAction**

*B_acceptAction* : T_algOp → T_boolean
Example: *B_acceptAction*(*o*)(*p*)
Symbol:

Returns **true** if the OAPT *p* meets the criteria as defined for the cost-controlled heuristic search strategy. The goal of this additional criteria is to keep the OAPT with the lowest cost as the current OAPT.

**initSS**

*B_initSS* : T_algOp
Example: *B_initSS*(*o*)
Symbol:

Returns the initial OAPT of the search space from where the search strategy object *o* will start the search. As a side effect, it initializes *B_current* (*B_setCurrent*), *B_transfRules*, and *B_currCost*.

**stopCond**

*B_stopCond* : T_boolean
Example: *B_stopCond*(*o*)
Symbol:

Returns the object **true** if the condition given to stop the search process *o* holds. Otherwise **false** is returned.

**setNextState**

*B_setNextState* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_setNextState*(*o*)(*p*)
Symbol:

Returns the next state in the search space to be applied and action on. It determines in which way the states are investigated in the search space.

**action**

*B_action* : T_algOp → T_list⟨T_algOp⟩
Example: *B_action*(*o*)(*p*)
Symbol:

Generates a list of successor states for the state *p* by applying algebraic equivalence transformation rules on *p* that are chosen according to heuristics defined for the search.

**argTypes**

*B_argTypes* : T_list⟨T_type⟩
Example: *B_argTypes*(*o*)
Symbol:

Returns a list of types which denote the types and ordering of the argument objects for the cost-controlled heuristic search strategy object *o*.

**resultType**
*B_resultType* : `T_type`
Example: *B_resultType*($o$)
Symbol:

| Returns the result type of the cost-controlled heuristc search strategy object $o$. |
|---|

**source**
*B_source* : `T_object`
Example: *B_source*($o$)
Symbol:

| Returns the source code of the cost-controlled heuristic search strategy object $o$. |
|---|

**compile**
*B_compile* : `T_object`
Example: *B_compile*($o$)
Symbol:

| Compiles the cost-controlled heuristic search strategy object $o$ and produces executable code which is stored in *B_executable*. |
|---|

**executable**
*B_executable* : `T_object`
Example: *B_executable*($o$)
Symbol:

| Returns the executable code of the cost-controlled heuristic search strategy object $o$. |
|---|

**execute**
*B_execute* : `T_list`⟨`T_algOp`⟩ → `T_algOp`
Example: *B_execute*($o$)($p$)
Symbol:

| Executes the cost-controlled heuristic search strategy object $o$ using the list of OAPT objects $p$ as arguments and returns an "optimal" OAPT object. One requirement is that the list of arguments $p$ must be compatible with the argument type list for the function. |
|---|

**basicExecute**
*B_basicExecute* : `T_list`⟨`T_algOp`⟩ → `T_algOp`
Example: *B_basicExecute*($o$)($p$)
Symbol:

| It works the same as *B_execute*. |
|---|

**costFunction**
*B_costFunction* : `T_list`⟨`T_object`⟩ → `T_function`
Example: *B_costFunction*($o$)($p$)
Symbol:

| It returns a cost function object that when is executed returns a pre-estimated cost of executing the cost-controlled heuristic search strategy object $o$ with arguments $p$. |
|---|

Native Behaviors: Related to Query Optimization.

**currCost**
*B_currCost* : `T_integer`
Example: *B_currCost*($o$)
Symbol:

| Returns the cost of the current state of the search space that is being explored by the cost-controlled heuristic search strategy $o$. |
|---|

# T_enumSS

Supertypes: T_searchStrat
Subtypes: T_SystemR, T_AugHeur
Inherited Behaviors:

**goal**
*B_goal* : T_collection⟨T_algOp⟩
Example: *B_goal*(*o*)
Symbol:

| Returns the collection of states that have been chosen as "good" candidates for being returned as the optimized OAPT. |
|---|

**optimal**
*B_optimal* : T_algOp
Example: *B_optimal*(*o*)
Symbol:

| Returns the "optimal" OAPT from the collection *B_goal*. |
|---|

Overriden Behaviors: Replacement for Query Optimization purposes.

**initSS**
*B_initSS* : T_algOp
Example: *B_initSS*(*o*)
Symbol:

| Returns the initial state of the search space from where the search strategy object *o* will start the search. |
|---|

**stopCond**
*B_stopCond* : T_boolean
Example: *B_stopCond*(*o*)
Symbol:

| Returns the object **true** if the condition given to stop the search process *o* holds. Otherwise **false** is returned. |
|---|

**setNextState**
*B_setNextState* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_setNextState*(*o*)(*p*)
Symbol:

| Returns the next state in the search space to be applied and action on. It determines in which way the states are investigated in the search space. If its implementation chooses the least recent state, then the search strategy is breadth-first; if it chooses the most recently generated state, then it implements depth-first search. |
|---|

**action**
*B_action* : T_algOp → T_list⟨T_algOp⟩
Example: *B_action*(*o*)(*p*)
Symbol:

| Generates a list of successor states for the state *p* by applying algebraic equivalence transformation rules on it. |
|---|

**argTypes**
*B_argTypes* : T_list⟨T_type⟩
Example: *B_argTypes*(*o*)
Symbol:

| Returns a list of types which denote the types and ordering of the argument objects for the enumerative search strategy object *o*. |
|---|

**resultType**
*B_resultType* : T_type
Example: *B_resultType*(*o*)
Symbol:

| Returns the result type of the search strategy object *o*. |
|---|

**source**
*B_source* : T_object
Example: *B_source*(*o*)
Symbol:

| Returns the source code of the enumerative search strategy object *o*. |
|---|

**compile**
*B_compile* : T_object
Example: *B_compile*(*o*)
Symbol:

| Compiles the enumerative search strategy object *o* and produces executable code which is stored in *B_executable*. |
|---|

**executable**
*B_executable* : T_object
Example: *B_executable*(*o*)
Symbol:

| Returns the executable code of the enumerative search strategy object *o*. |
|---|

**execute**
*B_execute* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_execute*(*o*)(*p*)
Symbol:

| Executes the enumerative search strategy object *o* using the list of OAPT objects *p* as arguments and returns an "optimal" OAPT object. One requirement is that the list of arguments *p* must be compatible with the argument type list for the function. |
|---|

**basicExecute**
*B_basicExecute* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_basicExecute*(*o*)(*p*)
Symbol:

| It works the same as *B_execute*. |
|---|

**costFunction**
*B_costFunction* : T_list⟨T_object⟩ → T_function
Example: *B_costFunction*(*o*)(*p*)
Symbol:

| It returns a cost function object that when is executed returns a pre-estimated cost of executing the enumerativve search strategy object *o* with arguments *p*. |
|---|

Native Behaviors: Related to Query Optimization.

**prune**
*B_prune* : T_list⟨algOp⟩ → T_list⟨T_algOp⟩
Example: *B_prune*(*o*)(*p*)
Symbol:

| Returns the list of states that results after discarding some "bad" states from the list *p* of OAPT objects that was generated by the behavior *B_action*. |
|---|

**open**

*B_open* : T_list⟨T_algOp⟩
Example: *B_open*(*o*)
Symbol:

Returns the open list of OAPTs for the enumerative search strategy *o*. These OAPTs are candidate states to be explored by the search strategy *o*. (It is implemented by a stored function).

**current**

*B_current* : T_algOp
Example: *B_current*(*o*)
Symbol:

Returns the current state of the search space that is being explored by the search strategy *o*. It is generated by the behavior *B_setNextState*. (It is implemented by a stored function).

# T_randomSS

Supertypes: T_searchStrat
Subtypes: T_II, T_SA
Inherited Behaviors:

**goal**

*B_goal* : T_collection⟨T_algOp⟩
Example: *B_goal*(*o*)
Symbol:

| |
|---|
| Returns the collection of states that have been chosen as "good" candidates for being returned as the optimized OAPT. |

**optimal**

*B_optimal* : T_algOp
Example: *B_optimal*(*o*)
Symbol:

| |
|---|
| Returns the "optimal" OAPT from the collection *B_goal*. |

Overriden Behaviors: Replacement for Query Optimization purposes.

**initSS**

*B_initSS* : T_collection⟨T_algOp⟩
Example: *B_initSS*(*o*)(*p*)
Symbol:

| |
|---|
| Returns the of initial state(s) of the search space that the search strategy *o* uses to start the search process For example, while II is characterized by the choice of several start states, SA has only one initial state. |

**stopCond**

*B_stopCond* : T_boolean
Example: *B_stopCond*(*o*)
Symbol:

| |
|---|
| Returns the object **true** if the condition given to stop the search process *o* holds. Otherwise **false** is returned. It refers to the global stop condition for the randomized search strategy *o* |

**setNextState**

*B_setNextState* : T_list⟨T_algOp⟩ → T_algOp
Example: *B_setNextState*(*o*)(*p*)
Symbol:

| |
|---|
| Returns the next state in the search space to be applied and action on. It determines in which way the states are investigated in the search space. If its implementation generates a new OAPT object, then the search strategy is Iterative Improvement; if it changes the temperature parameter, then it implements Simulated Annealing |

**action**

*B_action* : T_algOp → T_list⟨T_algOp⟩
Example: *B_action*(*o*)(*p*)
Symbol:

| |
|---|
| Generates a list of neighbor states for the state *p* by applying algebraic equivalence transformation rules to the complete OAPT *p*. Each of the generated neighbors is a complete OAPT. |

**argTypes**

*B_argTypes* : `T_list⟨T_type⟩`
Example: *B_argTypes*(*o*)
Symbol:

| Returns a list of types which denote the types and ordering of the argument objects for the randomized search strategy object *o*. |
|---|

**resultType**

*B_resultType* : `T_type`
Example: *B_resultType*(*o*)
Symbol:

| Returns the result type of the search strategy object *o*. |
|---|

**source**

*B_source* : `T_object`
Example: *B_source*(*o*)
Symbol:

| Returns the source code of the randomized search strategy object *o*. |
|---|

**compile**

*B_compile* : `T_object`
Example: *B_compile*(*o*)
Symbol:

| Compiles the randomized search strategy object *o* and produces executable code which is stored in *B_executable*. |
|---|

**executable**

*B_executable* : `T_object`
Example: *B_executable*(*o*)
Symbol:

| Returns the executable code of the randomized search strategy object *o*. |
|---|

**execute**

*B_execute* : `T_list⟨T_algOp⟩` → `T_algOp`
Example: *B_execute*(*o*)(*p*)
Symbol:
Derivation:
Executes the randomized search strategy object *o* using the list of OAPT objects *p* as arguments and returns an "optimal" OAPT object. One requirement is that the list of arguments *p* must be compatible with the argument type list for the function.

□

**basicExecute**

*B_basicExecute* : `T_list⟨T_algOp⟩` → `T_algOp`
Example: *B_basicExecute*(*o*)(*p*)
Symbol:

| It works the same as *B_execute*. |
|---|

**costFunction**

*B_costFunction* : `T_list⟨T_object⟩` → `T_function`
Example: *B_costFunction*(*o*)(*p*)
Symbol:

| It returns a cost function object that when is executed returns a pre-estimated cost of executing the randomized search strategy object *o* with arguments *p*. |
|---|

Native Behaviors: Related to Query Optimization.

**localStopCond**

*B_localStopCond* : `T_boolean`
Example: *B_localStopCond*(*o*)
Symbol:

| |
|---|
| Returns the object **true** when a local minimum has been found. Otherwise **false** is returned. It refers to the local stop condition for the randomized search strategy *o* (i.e. elapsed time for II, temperature for SA, and so on) |

**acceptAction**

*B_acceptAction* : `T_list`⟨`algOp`⟩ → `T_boolean`
Example: *B_acceptAction*(*o*)(*p*)
Symbol:

| |
|---|
| Returns **true** if the criterion for accepting a transformation is satisfied by the transformed OAPTs. |

**nmoves**

*B_nmoves* : `T_integer`
Example: *B_nmoves*(*o*)
Symbol:

| |
|---|
| Returns the number of transformations that have been applied to the current state, when searching for its local minimum. It is implemented by a stored function. |

**currState**

*B_currState* : `T_collection`⟨`T_algOp`⟩
Example: *B_currState*(*o*)
Symbol:

| |
|---|
| Returns the current state(s) of the search space that is being explored by the randomized search strategy *o*. It is generated by the behavior *B_setNextState*. (It is implemented by a stored function). |